Für PowerShell 2 bis 6

holger SCHWICHTENBERG

WINDOWS

# PowerShell 5.1 und PowerShell Core 6.1

3. Auflage

## DAS PRAXISBUCH

»Sehr gut« dotnetpro zur 2. Auflage

Im Internet: Codebeispiele, Forum, PowerShell-Kurzreferenz

HANSER

www.IT-Visions.de
Dr. Holger Schwichtenberg

Schwichtenberg

Windows PowerShell 5.1 und PowerShell Core 6.1

Holger Schwichtenberg

# Windows PowerShell 5.1 und PowerShell Core 6.1

## Das Praxisbuch

3., aktualisierte Auflage

HANSER

Der Autor:
*Dr. Holger Schwichtenberg,* Essen
www.IT-Visions.de




Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.


Lektorat: Sylvia Hasselbach
Copy editing: Sandra Gottmann, Nienberge
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Stephan Rönigk
Gesamtherstellung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

E-Book-ISBN: 978-3-446-45923-6

# Inhalt

# Vorwort

Liebe Leserin, lieber Leser,

willkommen zur aktuellen Auflage des PowerShell-Buchs! Es handelt sich hierbei um die dritte Auflage des Windows PowerShell 5-Buches und die siebte Auflage des PowerShell-Buches insgesamt, das erstmalig 2007 bei Addison-Wesley erschienen ist. Das vor Ihnen liegende Buch behandelt die Windows PowerShell in der Version 5.1 sowie die PowerShell Core in der Version 6.0/6.1 von Microsoft sowie ergänzende Werkzeuge von Microsoft und Drittanbietern (z. B. PowerShell Community Extensions). Das Buch ist aber auch geeignet, wenn Sie noch PowerShell 2.0, 3.0, 4.0 oder 5.0 einsetzen. Welche Funktionen neu hinzugekommen sind, wird jeweils erwähnt.

## Wer bin ich?

Mein Name ist Holger Schwichtenberg, ich bin derzeit 45 Jahre alt und habe im Fachgebiet Wirtschaftsinformatik promoviert. Ich lebe (in Essen, im Herzen des Ruhrgebiets) davon, dass mein Team und ich im Rahmen unserer Firma *www.IT-Visions.de* anderen Unternehmen bei der Entwicklung von .NET-, Web- und PowerShell-Anwendungen beratend und schulend zur Seite stehen. Zudem entwickeln wir im Rahmen der 5Minds IT-Solutions GmbH & Co. KG Software (*www.5Minds.de*) im Auftrag von Kunden in zahlreichen Branchen.

Es ist mein Hobby und Nebenberuf, IT-Fachbücher zu schreiben. Dieses Buch ist, unter Mitzählung aller nennenswerten Neuauflagen, das 68. Buch, das ich allein oder mit Co-Autoren geschrieben habe. Meine weiteren Hobbys sind Mountain Biking, Lauf-Sport, Fotografie und Reisen.

Natürlich verstehe ich das Bücherschreiben auch als Werbung für die Arbeit unserer Unternehmen und wir hoffen, dass der ein oder andere von Ihnen uns beauftragen wird, Ihre Organisation durch Beratung, Schulung und Auftragsentwicklung zu unterstützen.

## Wer sind Sie?

Damit Sie den optimalen Nutzen aus diesem Buch ziehen können, möchte ich – so genau es mir möglich ist – beschreiben, an wen sich dieses Buch richtet. Hierzu habe ich einen Fragebogen ausgearbeitet, mit dem Sie schnell erkennen können, ob das Buch für Sie geeignet ist.

| | | |
|---|---|---|
| Sind Sie Systemadministrator in einem Windows-Netzwerk? | ○ Ja | ○ Nein |
| Laufen die für Sie relevanten Computer mit den von PowerShell 3.0, 4.0, 5.x oder 6.x unterstützten Betriebssystemen? (Windows 7/8/8.1/10, Windows Server 2008/2008 R2/2012/2012 R2/2016)<br>Hinweis: Die PowerShell Core 6.0 für Linux und MacOS wird nur als Randthema kurz in diesem Buch behandelt, da es hier bislang kaum Befehle für die PowerShell gibt! | ○ Ja | ○ Nein |
| Sie besitzen zumindest rudimentäre Grundkenntnisse im Bereich des (objektorientierten) Programmierens? | ○ Ja | ○ Nein |
| Wünschen Sie einen kompakten Überblick über die Architektur, Konzepte und Anwendungsfälle der PowerShell? | ○ Ja | ○ Nein |
| Sie können auf Schritt-für-Schritt-Anleitungen verzichten? | ○ Ja | ○ Nein |
| Sie können auf formale Syntaxbeschreibungen verzichten und lernen lieber an aussagekräftigen Beispielen? | ○ Ja | ○ Nein |
| Sie erwarten nicht, dass in diesem Buch **alle** Möglichkeiten der PowerShell detailliert beschrieben werden? | ○ Ja | ○ Nein |
| Sind Sie, nachdem Sie ein Grundverständnis durch dieses Buch gewonnen haben, bereit, Detailfragen in der Dokumentation der PowerShell, von .NET und WMI nachzuschlagen, da das Buch auf 1200 Seiten nicht alle Details erläutern kann? | ○ Ja | ○ Nein |

Wenn Sie alle obigen Fragen mit „Ja“ beantwortet haben, ist das Buch richtig für Sie. In anderen Fällen sollten Sie sich erst mit einführender Literatur beschäftigen.

Die PowerShell Core 6.0, die zum letzten Redaktionsschluss noch auf dem Beta-Stand war, ist inzwischen erschienen. Das Buch wurde auf die RTM-Version 6.0 sowie die Release Candidate-Version von PowerShell Core 6.1 aktualisiert. Zudem wurde ein Unterkapitel zur Installation von Windows-Features in Client-Betriebssystemen im Kapitel „Teil C/Softwareverwaltung“ sowie ein Praxisbeispiel zur Installation des Webservers „IIS“ ergänzt. Außerdem wurde das Kapitel „Verwendung von .NET-Klassen“ erweitert.

## Was ist neu in diesem Buch?

Gegenüber der vorherigen Auflage zur PowerShell 5.0 wurde das Buch um die neuen Funktionen in Windows PowerShell 5.1 sowie PowerShell Core 6.0 erweitert und inhaltlich optimiert. Praxiseinsatzkapitel wurden ergänzt zu Windows Update, Windows Nano Server und Docker-Containern. Zudem wurden die bestehenden Inhalte des Buchs an vielen Stellen erweitert und didaktisch optimiert.

## Sind in diesem Buch alle Features der PowerShell beschrieben?

Die PowerShell umfasst mittlerweile über 1500 Commandlets mit jeweils zahlreichen Optionen. Zudem gibt es unzählige Erweiterungen mit vielen hundert weiteren Commandlets. Zudem existieren zahlreiche Zusatzwerkzeuge. Es ist allein schon aufgrund der Vorgaben des Verlags für den Umfang des Buchs nicht möglich, alle Commandlets und Parameter hier auch nur zu erwähnen. Zudem habe ich – obwohl ich selbst fast jede Woche mit der PowerShell in der Praxis arbeite – immer noch nicht alle Commandlets und alle Parameter jemals eingesetzt. Ich beschreibe in diesem Buch, was ich selbst in der Praxis, in meinen Schulungen und bei Kundeneinsätzen verwende. Es macht auch keinen Sinn, jedes Detail der PowerShell hier zu dokumentieren. Stattdessen gebe ich Ihnen **Hilfe zur Selbsthilfe**, damit Sie die Konzepte gut verstehen und sich dann Sonderfälle selbst erarbeiten können.

## Wie aktuell ist dieses Buch?

Die Informationstechnik hat sich immer schon schnell verändert. Seit aber auch Microsoft das Themen „Agilität“ und „Open Source“ für sich entdeckt hat, ist die Entwicklung nicht mehr schnell, sondern zum Teil rasant:

- Es erscheinen in kurzer Abfolge immer neue Produkte.
- Produkte erscheinen schon in frühen Produktstadien als „Preview“ mit Versionsnummern wie 0.1.
- Produkte ändern sich häufig. Aufwärts- und Abwärtskompatibilität ist kein Ziel mehr. Es wird erwartet, dass Sie Ihre Lösungen ständig den neuen Gegebenheiten anpassen.
- Produkte werden nicht mehr so ausführlich dokumentiert wie früher. Teilweise erscheint Dokumentation erst deutlich nach dem Erscheinen der Software.
- Produkte werden schnell auch wieder abgekündigt, wenn sie sich aus der Sicht der Hersteller bzw. aufgrund des Nutzerfeedbacks nicht bewährt haben.

Unter diesen neuen Einflussströmen steht natürlich auch dieses etablierte Buch. Leider kann man ein Buch nicht so schnell ändern wie Software.

Daher kann es passieren, dass – auch schon kurz nach dem Erscheinen dieses Buchs – einzelne Informationen in diesem Buch nicht mehr zu neueren Versionen passen. Wenn Sie so einen Fall feststellen, schreiben Sie bitte eine Nachricht an mich im Leser-Portal (siehe unten). Ich werde dies dann in Neuauflagen des Buchs berücksichtigen.

## Wem ist zu danken?

Folgenden Personen möchte ich meinen Dank für ihre Mitwirkung an diesem Buch aussprechen:

- meinem Kollegen und Freund Peter Monadjemi, der rund 100 Seiten mit Beispielen zu der Vor-Vor-Vor-Auflage dieses Buchs beigetragen hat (Themen: Workflows, Bitlocker, ODBC, Hyper-V, DNS-Client, Firewall und SQL-Server-Administration),
- Frau Sylvia Hasselbach, die mich schon seit 20 Jahren als Lektorin begleitet und die dieses Buchprojekt beim Carl Hanser Verlag koordiniert und vermarktet,
- Frau Sandra Gottmann, die meine Tippfehler gefunden und sprachliche Ungenauigkeiten eliminiert hat,
- meiner Frau und meinen Kindern dafür, dass sie mir das Umfeld geben, um neben meinem Hauptberuf an Büchern wie diesem zu arbeiten.

## Woher bekommen Sie die Beispiele aus diesem Buch?

Unter *http://www.powershell-doktor.de/leser* biete ich ein **ehrenamtlich betriebenes** Webportal für Leser meiner Bücher an. In diesem Portal können Sie

- die Codebeispiele aus diesem Buch in einem Archiv herunterladen,
- eine PowerShell-Kurzreferenz „Cheat Sheet“ (zwei DIN-A4-Seiten als Hilfe für die tägliche Arbeit) kostenlos herunterladen,
- Feedback zu diesem Buch geben (Bewertung abgeben und Fehler melden) und
- technische Fragen in einem Webforum stellen.

Alle registrierten Leser erhalten auch Einladungen zu kostenlosen Community-Veranstaltungen sowie Vergünstigungen bei unseren öffentlichen Seminaren zu .NET und zur PowerShell. Bei der Registrierung müssen Sie das Kennwort **Rogue One** angeben.

## Wie sind die Programmcodebeispiele organisiert?

Die Beispiele sind im Archiv organisiert nach den Buchteilen und innerhalb der Buchteile nach Kapitelnamen (verkürzt). In diesem Buch wird für den Zugriff auf die Beispieldateien das X:-Laufwerk verwendet. Dies müssen Sie auf Ihre Situation anpassen!

```
PS T:\> dir x:\


    Verzeichnis: x:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-r---        29.06.2017     23:56                1_Basiswissen
d-r---        28.06.2017     17:09                2_Aufbauwissen
d-r---        02.06.2017     10:38                3_Einsatzgebiete
d-r---        30.06.2017     17:22                4_Profiwissen
```

```
PS T:\> dir x:\1_Basiswissen\


    Verzeichnis: x:\1_Basiswissen


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        29.06.2017     23:56                Aliase
d-r---        24.04.2017     09:52                Ausgaben
d-r---        30.05.2017     00:28                Commandlets
d-----        26.06.2017     10:40                ErsteSchritte
d-r---        29.06.2017     23:34                Hilfe
d-----        30.05.2017     20:59                Module
d-r---        26.03.2014     12:49                Navigation
d-r---        04.06.2017     11:21                Pipelining
d-----        30.05.2017     21:15                PowerShellLanguage
d-----        29.05.2017     23:57                PowerShellOOP
d-----        30.06.2017     18:47                PSCore
d-r---        30.05.2017     20:46                Scripting
d-r---        26.03.2014     12:49                TippsAndTricks
d-r---        26.03.2014     12:49                Werkzeuge
d-r---        26.03.2014     12:49                WPS versus VBS
d-----        03.05.2016     14:12                Zeichenkettenbearbeitung
```

## Wo können Sie sich schulen oder beraten lassen?

Unter der E-Mail-Adresse *kundenteam@IT-Visions.de* stehen mein Team und ich für Anfragen bezüglich Schulung, Beratung und Entwicklungstätigkeiten zur Verfügung – nicht nur zum Thema PowerShell und .NET/.NET Core, sondern zu fast allen modernen Techniken der Entwicklung und des Betriebs von Software in großen Unternehmen. Wir besuchen Sie gerne in Ihrem Unternehmen an einem beliebigen Standort.

## Zum Schluss des Vorworts …

… wünsche ich Ihnen viel Spaß und Erfolg mit der PowerShell!

*Dr. Holger Schwichtenberg*

*Essen, im Oktober 2018*

# Über den Autor Dr. Holger Schwichtenberg

- Studienabschluss Diplom-Wirtschaftsinformatik an der Universität Essen
- Promotion an der Universität Essen im Gebiet komponentenbasierter Softwareentwicklung
- Seit 1996 selbstständig als unabhängiger Berater, Dozent, Softwarearchitekt und Fachjournalist
- Leiter des Berater- und Dozententeams bei *www.IT-Visions.de*

www.IT-Visions.de®
Dr. Holger Schwichtenberg

- Leitung der Softwareentwicklung im Bereich Microsoft/.NET bei der 5Minds IT-Solutions GmbH & Co. KG (*www.5minds.de*)

5Minds
IT-SOLUTIONS

- Über 65 Fachbücher beim Carl Hanser Verlag, bei O'Reilly, Microsoft Press und Addison-Wesley sowie mehr als 950 Beiträge in Fachzeitschriften
- Gutachter in den Wettbewerbsverfahren der EU gegen Microsoft (2006–2009)
- Ständiger Mitarbeiter der Zeitschriften iX (seit 1999), dotnetpro (seit 2000) und Windows Developer (seit 2010) sowie beim Online-Portal *heise.de* (seit 2008)
- Regelmäßiger Sprecher auf nationalen und internationalen Fachkonferenzen (z. B. Microsoft TechEd, Microsoft Summit, Microsoft IT Forum, BASTA, BASTA-on-Tour, .NET Architecture Camp, Advanced Developers Conference, Developer Week, OOP, DOTNET Cologne, MD DevDays, Community in Motion, DOTNET-Konferenz, VS One, NRW.Conf, Net.Object Days, Windows Forum)Zertifikate und Auszeichnungen von Microsoft:
  - Bereits 14 mal ausgezeichnet als Microsoft Most Valuable Professional (MVP)
  - Zertifiziert als Microsoft Certified Solution Developer (MCSD)
- Thematische Schwerpunkte:
  - Softwarearchitektur, mehrschichtige Softwareentwicklung, Softwarekomponenten, SOA

  - Microsoft .NET Framework, Visual Studio, C#, Visual Basic
  - .NET-Architektur/Auswahl von .NET-Technologien
  - Einführung von .NET Framework und Visual Studio/Migration auf .NET
  - Webanwendungsentwicklung und Cross-Plattform-Anwendungen mit HTML, ASP.NET, JavaScript/TypeScript und Webframeworks wie Angular
  - Enterprise .NET, verteilte Systeme/Webservices mit .NET insbes. Windows Communication Foundation und WebAPI
  - Relationale Datenbanken, XML, Datenzugriffsstrategien
  - Objektrelationales Mapping (ORM), insbesondere ADO.NET Entity Framework und EF Core
  - Windows PowerShell, PowerShell Core und Windows Management Instrumentation (WMI)
- Ehrenamtliche Community-Tätigkeiten:
  - Vortragender für die International .NET Association (INETA)
  - Betrieb diverser Community-Websites: *www.dotnetframework.de*, *www.entwicklerlexikon.de*, *www.windows-scripting.de*, *www.aspnetdev.de* u. a.
- Firmenwebsites: *http://www.IT-Visions.de* und *http://www.5minds.de*
- Weblog: *http://www.dotnet-doktor.de*
- Kontakt: *kundenteam@IT-Visions.de* sowie *Telefon 02 01-64 95 90-0*

# A

# Teil A: PowerShell-Basiswissen

Dieser Buchteil informiert über die Basiskonzepte der PowerShell, insbesondere Commandlets, Pipelines, Navigation und Skripte. Außerdem werden am Ende dieses Teils Werkzeuge vorgestellt.

# 1 Erste Schritte mit der PowerShell

Das DOS-ähnliche Kommandozeilenfenster hat viele Windows-Versionen in beinahe unveränderter Form überlebt. Mit der Windows PowerShell (WPS) besitzt Microsoft seit dem Jahr 2006 einen Nachfolger, der es mit den Unix-Shells aufnehmen kann und diese in Hinblick auf Eleganz und Robustheit in einigen Punkten auch überbieten kann. Die PowerShell ist eine Adaption des Konzepts von Unix-Shells auf Windows unter Verwendung des .NET Frameworks und mit Anbindung an die Windows Management Instrumentation (WMI). Seit dem Jahr 2017 gibt es die PowerShell auch für Linux und MacOS als „PowerShell Core".

## 1.1 Was ist die PowerShell?

In einem Satz: Die **Windows PowerShell (WPS)** ist eine .NET-basierte Umgebung für interaktive Systemadministration und Skripting auf der Windows-Plattform. **PowerShell Core (PS Core)** ist eine .NET Core-basierte Umgebung für interaktive Systemadministration und Skripting auf Windows, Linux und MacOS.

Die Kernfunktionen der PowerShell sind:

- Zahlreiche eingebaute Befehle, die „Commandlets" genannt werden
- Zugang zu allen Systemobjekten, die durch COM-Bibliotheken, das .NET Framework und die Windows Management Instrumentation (WMI) bereitgestellt werden
- Robuster Datenaustausch zwischen Commandlets durch Pipelines basierend auf typisierten Objekten
- Ein einheitliches Navigationsparadigma für verschiedene Speicher (z. B. Dateisystem, Registrierungsdatenbank, Zertifikatsspeicher, Active Directory und Umgebungsvariablen)
- Eine einfach zu erlernende, aber mächtige Skriptsprache mit wahlweise schwacher oder starker Typisierung
- Ein Sicherheitsmodell, das die Ausführung unerwünschter Skripte unterbindet
- Integrierte Funktionen für Ablaufverfolgung und Debugging
- Die PowerShell kann um eigene Befehle erweitert werden.
- Die PowerShell kann in eigene Anwendungen integriert werden (Hosting).

## 1.2 Windows PowerShell versus PowerShell Core

Die Windows PowerShell 5.1 ist weit mächtiger als die PowerShell Core 6.x, weil die PowerShell Core einen Neustart des PowerShell-Entwicklungsprojekts in Hinblick auf Plattformunabhängigkeit darstellt. In PowerShell Core fehlen viele Commandlets der Grundausstattung der Windows PowerShell und viele der verfügbaren PowerShell-Erweiterungsmodule laufen bisher nicht in der PowerShell Core.

Details zu den Funktionseinschränkungen der PowerShell Core lesen Sie im Kapitel „PowerShell Core 6.x für Windows, Linux und MacOS“.

**TIPP:** Wenn Sie unter Windows arbeiten, sollten Sie daher vorerst noch die Windows PowerShell (nach Möglichkeit in der aktuellen Version 5.1) verwenden.

Unter Linux und MacOS gibt es keine Windows PowerShell. Hier können Sie die PowerShell Core 6.x verwenden. Der Wert der PowerShell Core unter Linux und MacOS liegt in den mächtigen Pipelining- sowie Ein- und Ausgabe-Commandlets. Für konkrete Zugriffe auf das Betriebssystem gibt es hingegen für die PowerShell Core unter MacOS und Linux noch fast keine Commandlets. Man wird also hier immer klassische Linux- und MacOS-Kommandozeilenbefehle mit zeichenkettenbasierter Verarbeitung in die PowerShell einbinden. Wie dies geht, wird im Kapitel „PowerShell Core 6.x für Windows, Linux und MacOS“ erklärt.

## 1.3 Windows PowerShell herunterladen und auf anderen Windows-Betriebssystemen installieren

Die Windows PowerShell 5.1 ist in Windows 10 (ab Anniversary Update) und Windows Server 2016 bereits im Standard installiert.

Wenn Sie nicht Windows 10 oder Windows Server 2016 benutzen, müssen Sie die PowerShell 5.1 erst installieren.

Die nachträgliche Installation der Windows PowerShell 5.1 ist auf folgenden Betriebssystemen möglich:

- Windows Server 2012 R2
- Windows Server 2012
- Windows 2008 R2
- Windows 8.1
- Windows 7

Die Windows PowerShell 5.1 wird auf diesen Betriebssystemen als Teil des Windows Management Framework 5.1 (WMF) installiert – *https://www.microsoft.com/en-us/download/details.aspx?id=54616*.

Bei der Installation ist zu beachten, dass jeweils das .NET Framework 4.5.2 oder höher vorhanden sein muss. Auch mit .NET Framework 4.6.x und 4.7 funktioniert die PowerShell 5.1.

Das WMF-5.1-Installationspaket betrachtet sich als Update für Windows (KB3191566 für Windows 7 und Windows Server 2008 R2 bzw. KB3191564 für Windows 8.1 und Windows Server 2012 R2 sowie KB3191565 für Server 2012).


**Bild 1.1** Installationspaket für PowerShell 5.1 als Erweiterung

### Installationsordner

Die Windows PowerShell installiert sich in folgendes Verzeichnis: *%systemroot%\system32\WindowsPowerShell\V1.0* (für 32-Bit-Systeme).

**ACHTUNG:** Dabei ist das „V1.0“ im Pfad tatsächlich richtig: Microsoft hat dies seit Version 1.0 nicht verändert. Geplant war wohl eine „Side-by-Side“-Installationsoption wie beim .NET Framework. Doch später hat sich Microsoft dann entschieden, dass eine neue PowerShell immer die alte überschreibt.

Auf 64-Bit-Systemen gibt es die PowerShell zweimal, einmal als 64-BitVersion in *%systemroot%\system32\WindowsPowerShell\V1.0* und einmal als 32-Bit-Version. Letztere findet man unter *%systemroot%\Syswow64\WindowsPowerShell\V1.0*. Die 32-Bit-Version braucht man, wenn man eine Bibliothek nutzen will, für die es keine 64Bit-Version gibt, z. B. den Zugriff auf Microsoft-Access-Datenbanken.

Es handelt sich auch dabei nicht um einen Tippfehler: Die 64-Bit-Version befindet sich in einem Verzeichnis, das „32“ im Namen trägt, und die 32-Bit-Version in einem Verzeichnis mit „64“ im Namen!

Die 32-Bit-Version der PowerShell und die 64-Bit-Version der PowerShell sieht man im Startmenü: Die 32-Bit-Version hat den Zusatz „(x86)“. Die 64-Bit-Version hat keinen Zusatz. Auch den Editor „ISE“ gibt es in einer 32- und einer 64-Bit-Version.

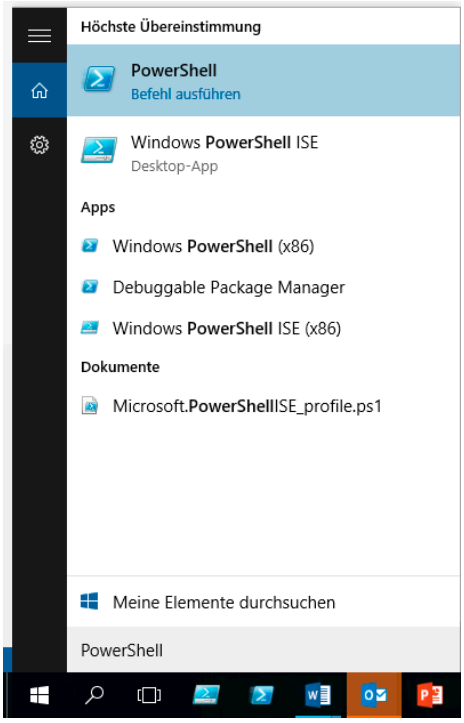



**Bild 1.2**
PowerShell-Einträge im Windows-10-Startmenü

**TIPP:** Unter Windows 8.x empfiehlt sich der Einsatz der Erweiterung *http://www.classicshell.net*, die das klassische Startmenü in Windows 8.x zurückbringt. Der Rückgriff auf ein Startmenü hat nicht nur mit Nostalgie zu tun, sondern auch ganz handfeste praktische Gründe: Der kachelbasierte Startbildschirm von Windows 8.x findet leider zum Suchbegriff „PowerShell" weder die PowerShell ISE noch die 32-Bit-Variante der PowerShell.

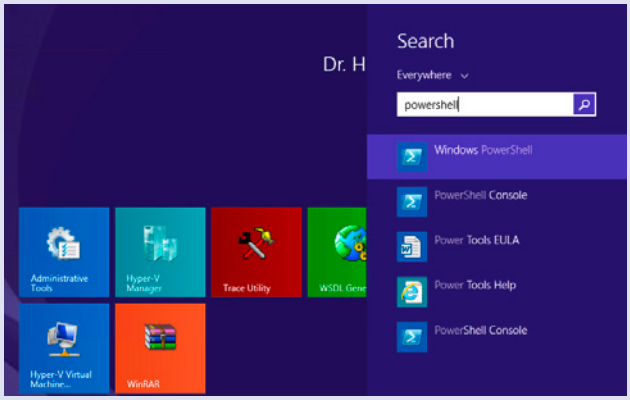



**Bild 1.3** Versagen auf ganzer Linie: Der kachelbasierte Startbildschirm von Windows 8.x findet leider zum Suchbegriff „PowerShell" weder die ISE noch die 32-Bit-Variante der PowerShell. In Windows 10 ist das behoben

Unter Windows 8.x geht es mit Classic Shell:

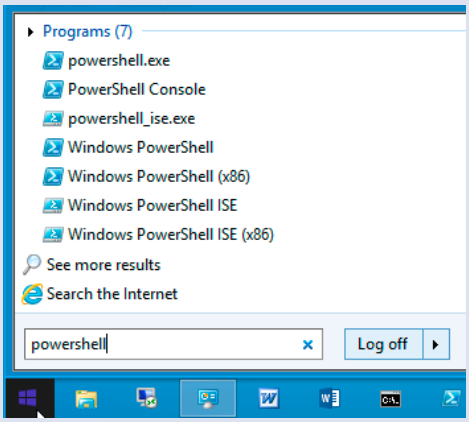



**Bild 1.4**
Die Classic Shell findet alle Einträge zur Windows PowerShell.

## Ereignisprotokoll „PowerShell“

Durch die Installation der PowerShell wird in Windows auch ein neues Ereignisprotokoll „PowerShell“ angelegt, in dem die PowerShell wichtige Zustandsänderungen der PowerShell protokolliert.

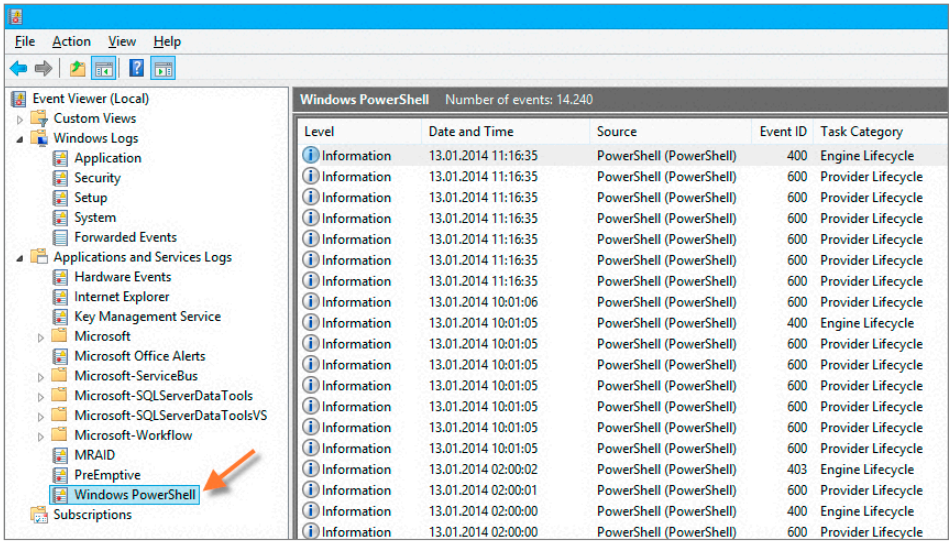



**Bild 1.5** Ereignisprotokoll „Windows PowerShell“

## Deinstallation

Falls man die PowerShell deinstallieren möchte, muss man dies in der Systemsteuerung unter „Programme und Funktionen/Installierte Updates anzeigen“ tun und dort das „Microsoft Windows Management Framework“ deinstallieren.

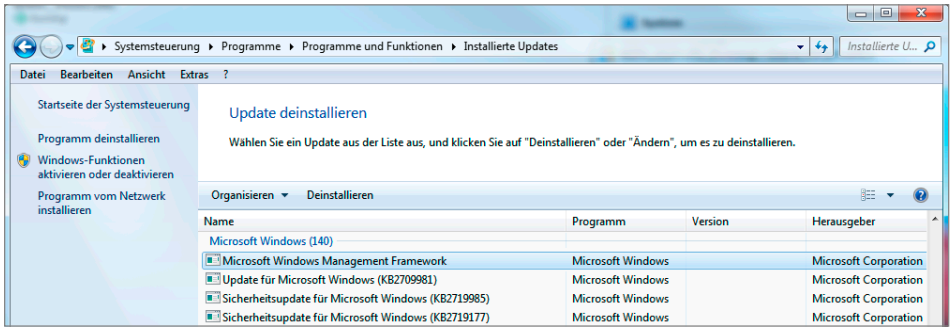



**Bild 1.6** Deinstallation der PowerShell durch Deinstallation des WMF

## 1.4 Die Windows PowerShell testen

Dieses Kapitel stellt einige Befehle vor, mit denen Sie die PowerShell-Funktionalität ausprobieren können. Die PowerShell verfügt über zwei Modi (interaktiver Modus und Skriptmodus), die hier getrennt behandelt werden.

### 1.4.1 PowerShell im interaktiven Modus

Der erste Test verwendet die PowerShell im interaktiven Modus.

Starten Sie bitte die PowerShell. Es erscheint ein leeres PowerShell-Konsolenfenster. Auf den ersten Blick ist kein großer Unterschied zur herkömmlichen Konsole zu erkennen. Allerdings steckt in der PowerShell mehr Kraft im wahrsten Sinne des Wortes.

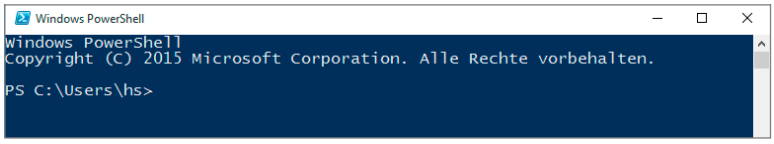



**Bild 1.7** Leeres PowerShell-Konsolenfenster

Geben Sie an der Eingabeaufforderung „Get-Process“ ein (wobei die Groß-/Kleinschreibung irrelevant ist. Das gilt nicht nur für Windows, sondern auch MacOS und Linux!) und drücken Sie dann die **Enter**-Taste. Es erscheint eine Liste aller Prozesse, die auf dem lokalen Computer laufen. Dies war Ihre erste Verwendung eines einfachen PowerShell-Commandlets.

**HINWEIS:** Beachten Sie bitte, dass die Groß-/Kleinschreibung keine Rolle spielt, da PowerShell keine Unterschiede zwischen groß- und kleingeschriebenen Commandlet-Namen macht.

Geben Sie an der Eingabeaufforderung „Get-Service i*“ ein. Jetzt erscheint eine Liste aller installierten Dienste auf Ihrem Computer, deren Namen mit dem Buchstaben „i“ beginnen. Hier haben Sie ein Commandlet mit Parametern verwendet.

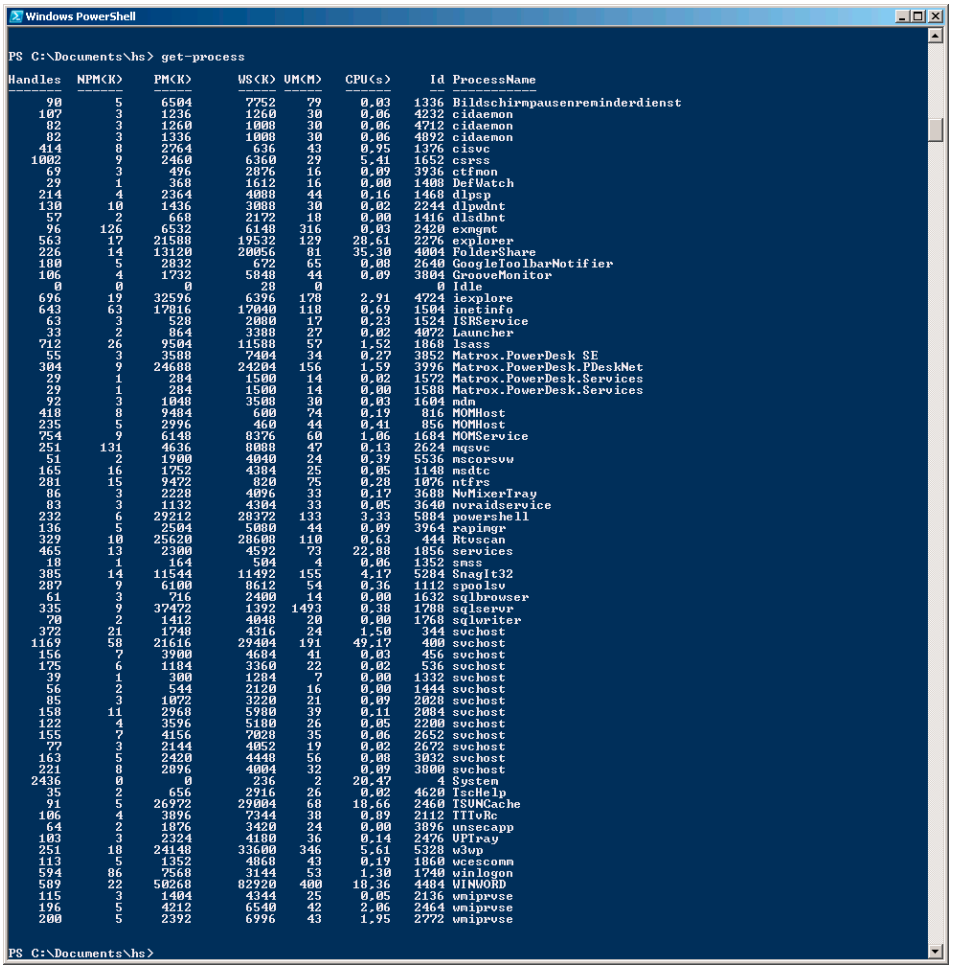



**Bild 1.8** Die Liste der Prozesse ist das Ergebnis nach Ausführung des Commandlets „Get-Process“.

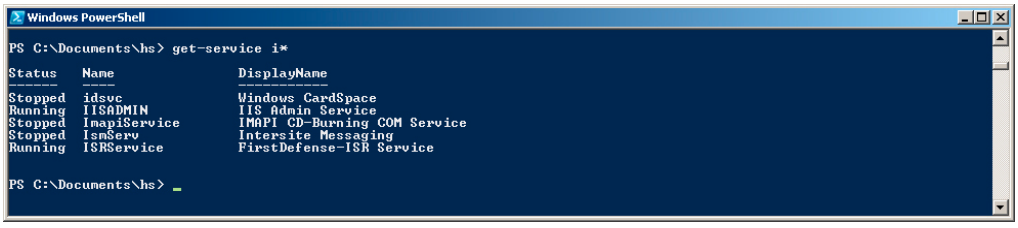



**Bild 1.9** Eine gefilterte Liste der Windows-Dienste

Geben Sie „Get-“ ein und drücken Sie dann mehrmals die **Tab**-Taste. Die PowerShell zeigt nacheinander alle Commandlets an, die mit dem Verb „get“ beginnen. Microsoft bezeichnet diese Funktionalität als „Tabulatorvervollständigung“. Halten Sie bei „Get-Eventlog“ an. Wenn Sie **Enter** drücken, fordert die PowerShell einen Parameter namens „LogName“ an. Bei „LogName“ handelt es sich um einen erforderlichen Parameter (Pflichtparameter). Nachdem Sie „Application“ eingetippt und die **Enter**-Taste gedrückt haben, erscheint eine lange Liste der aktuellen Einträge in Ihrem Anwendungsereignisprotokoll.

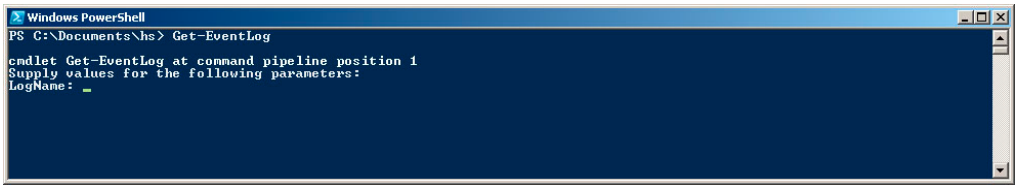



**Bild 1.10** PowerShell fragt einen erforderlichen Parameter ab.

Der letzte Test bezieht sich auf die Pipeline-Funktionalität der PowerShell. Auch geht es darum, die Listeneinträge aus dem Windows-Ereignisprotokoll aufzulisten, doch dieses Mal sind nur bestimmte Einträge interessant. Die Aufgabe besteht darin, die letzten zehn Ereignisse abzurufen, die sich auf das Drucken beziehen. Geben Sie den folgenden Befehl ein, der aus drei Commandlets besteht, die über Pipes miteinander verbunden sind:

```
Get-EventLog system | Where-Object { $_.source -eq "print" } | Select-Object -first 10
```

Die PowerShell scheint einige Sekunden zu hängen, nachdem die ersten zehn Einträge ausgegeben wurden. Dieses Verhalten ist korrekt, da das erste Commandlet (`Get-EventLog`) alle Einträge empfängt. Dieses Filtern geschieht durch aufeinanderfolgende Commandlets (`Where-Object` und `Select-Object`). Leider besitzt `Get-EventLog` keinen integrierten Filtermechanismus.

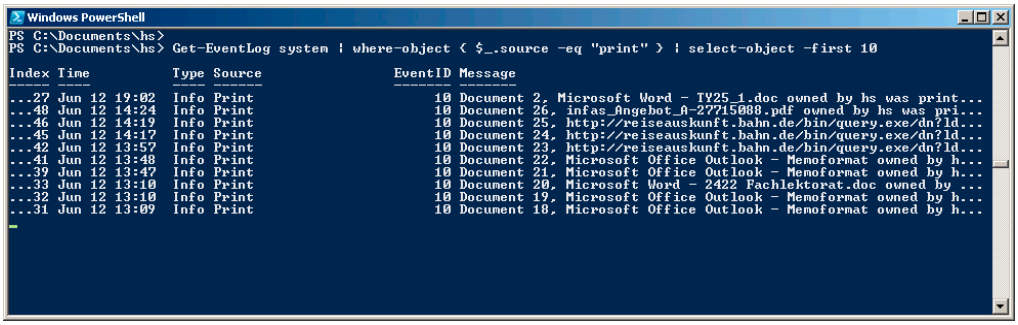



**Bild 1.11** Die Einträge des Ereignisprotokolls filtern

### 1.4.2 Installierte Version ermitteln

Die Windows PowerShell gibt bei ihrem Start ihre Versionsnummer nicht direkt preis. Nur die Jahreszahl im Copyright-Vermerk deutet indirekt auf die Versionsnummer hin. 2015 steht hier für die PowerShell 5.0, 2016 für die PowerShell 5.1. Die PowerShell Core meldet sich ohne Jahreszahl.

Die präzisere Versionsinformation ermittelt man durch den Abruf der eingebauten Variablen `$PSVersionTable`. Neben der PowerShell-Version erhält man auch Informationen über die Frameworks und Protokolle, auf denen die PowerShell aufsetzt.

Die „CLRVersion“ steht dabei für die Version der „Common Language Runtime“ (CLR), die Laufzeitumgebung des Microsoft .NET Framework. Es fehlt in der Versionstabelle leider die Information, dass die PowerShell 5.1 zwar mit der CLR-Version 4.0 zufrieden ist, aber die .NET-Klassenbibliothek in der Version 4.5.2 oder höher braucht, was eine Installation des .NET Frameworks 4.5.2 oder höher voraussetzt.

PowerShell Core 6.0 erfordert .NET Core 2.0. PowerShell Core 6.1 erfordert .NET Core 2.1. Allerdings braucht man .NET Core nicht separat installieren: Es wird beim Installationspaket von PowerShell Core mitgeliefert.

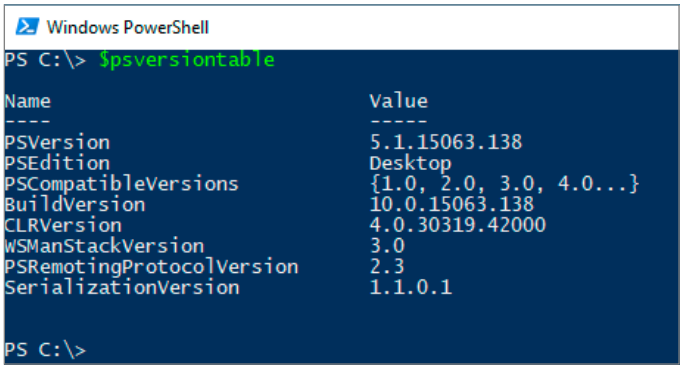



**Bild 1.12** Abruf der Versionsinformationen zur PowerShell 5.1 (hier unter Windows 10, Update-Stand 21. 04. 2017)

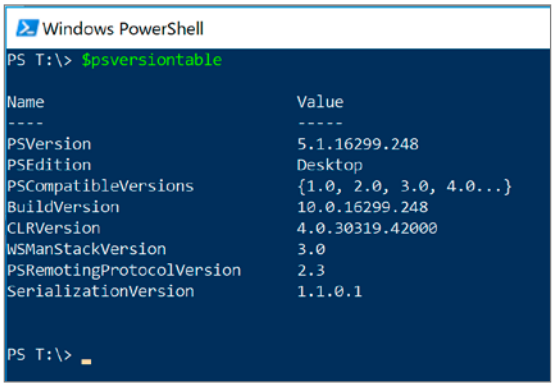



**Bild 1.13**
Abruf der Versionsinformationen zur PowerShell 5.1 (hier unter Windows 10, Update-Stand 02. 03. 2018)

Unter PowerShell Core hat Microsoft einige Anzeigen der Versionstabelle geändert. Am auffälligsten ist der Wert „Core“ statt „Desktop“ bei „PSEdition“ sowie die hinzugefügten Einträge „Platform“ und „OS“ für das aktuelle Betriebssystem. Platform hat die Werte Win32NT, MacOSX und Unix. Die „CLRVersion“ wird hier nicht mehr angezeigt. Microsoft verbirgt hier, welche Version von .NET Core bei PowerShell Core mitgeliefert wird.

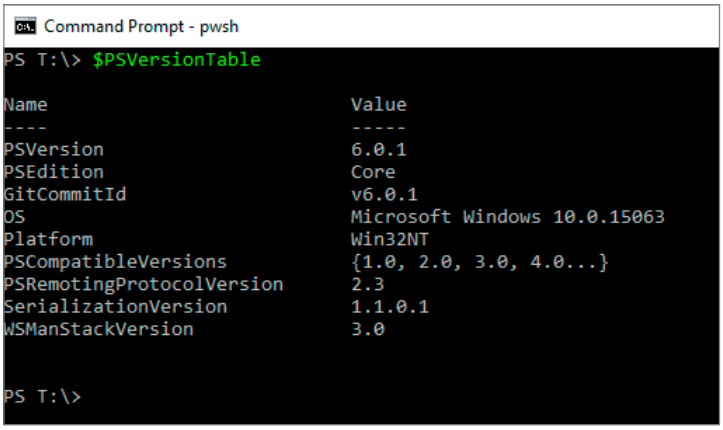



**Bild 1.14** Abruf der Versionsinformationen zur PowerShell Core 6.0.1 (hier unter Windows 10, Update-Stand 02. 03. 2018)

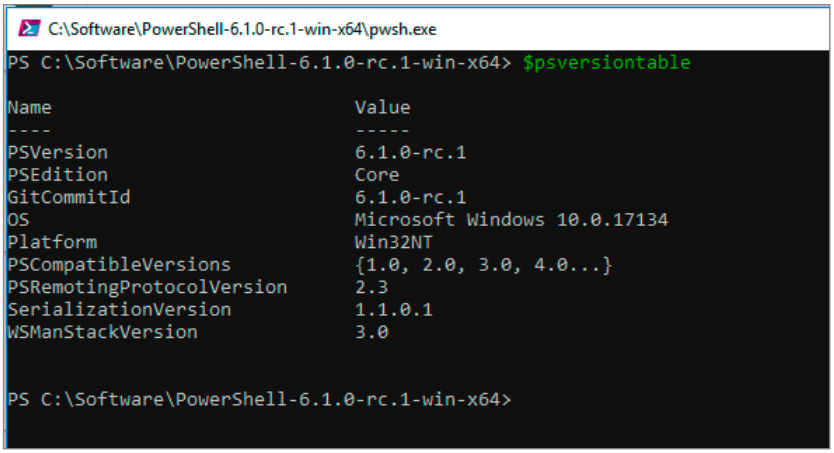



**Bild 1.15** Abruf der Versionsinformationen zur PowerShell Core 6.1 (hier unter Windows 10, Update-Stand 06. 09. 2018)

### 1.4.3 PowerShell im Skriptmodus

Bei einem PowerShell-Skript handelt es sich um eine Textdatei, die Commandlets und/oder Elemente der PowerShell-Skriptsprache (PSL) umfasst. Das zu erstellende Skript legt ein neues Benutzerkonto auf Ihrem lokalen Computer an.

### 1.4.4 Skript eingeben

Öffnen Sie den Windows-Editor „Notepad“ (oder einen anderen Texteditor) und geben Sie die folgenden Skriptcodezeilen ein, die aus Kommentaren, Variablendeklarationen, COM-Bibliotheksaufrufen und Shell-Ausgabe bestehen:

**Listing 1.1** Ein Benutzerkonto erstellen
[1_Basiswissen/ErsteSchritte/LocalUser_Create.ps1]

```
### PowerShell-Script
### Lokales Benutzerkonto anlegen
### (C) Holger Schwichtenberg

# Eingabewerte
$Name = "Dr. Holger Schwichtenberg"
$Accountname = "HolgerSchwichtenberg"
$Description = "Autor dieses Buchs / Website: www.powershell-doktor.de"
$Password = "secret+123"
$Computer = "localhost"

"Anlegen des Benutzerskonto $Name auf $Computer"

# Zugriff auf Container mit der COM-Bibliothek "Active Directory Service Interface"
$Container = [ADSI] "WinNT://$Computer"

# Benutzer anlegen
$objUser = $Container.Create("user", $Accountname)
$objUser.Put("Fullname", $Name)
$objUser.Put("Description", $Description)
# Kennwort setzen
$objUser.SetPassword($Password)
# Änderungen speichern
$objUser.SetInfo()

"Benutzer angelegt: $Name auf $Computer"
```

Speichern Sie die Textdatei unter dem Namen „createuser.ps1“ in einem Ordner auf der Festplatte, z. B. *x:\temp*. Beachten Sie, dass die Dateinamenserweiterung „.ps1“ lauten muss.

**HINWEIS:** Im Kapitel „Lokale Benutzer und Gruppen“ werden Sie lernen, dass es ab PowerShell 5.1 auch einen eleganteren Weg zum Anlegen lokaler Benutzer per Commandlet `New-LocalUser` gibt.

### 1.4.5 Skript starten

Starten Sie die PowerShell-Konsole. Versuchen Sie dort nun, das Skript zu starten. Geben Sie dazu

```
x:\temp\createuser.ps1
```

ein. Für die Ordner- und Dateinamen können Sie die Tabulatorvervollständigung verwenden! Der Versuch scheitert zunächst wahrscheinlich, da die Skriptausführung auf den meisten Windows-Betriebssystemversionen standardmäßig in der PowerShell nicht zulässig ist. Dies ist kein Fehler, sondern eine Sicherheitsfunktionalität. Denken Sie an den „Love Letter"-Wurm für den Windows Script Host!

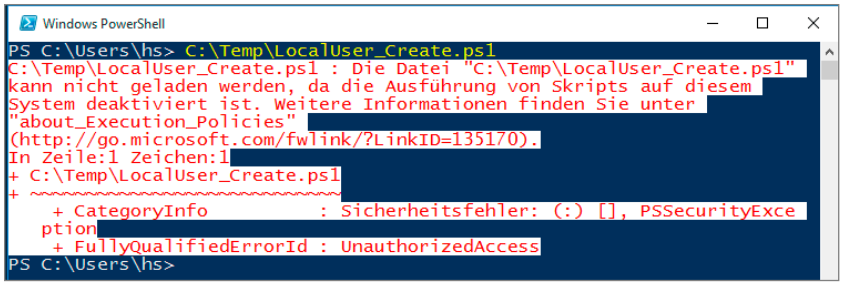



**Bild 1.16** Die Skriptausführung ist standardmäßig verboten.

**HINWEIS:** Bisher war die PowerShell-Skriptausführung auf allen Betriebssystemen im Standard verboten. Erstmals in Windows Server 2012 R2 hat Microsoft sie im Standard erlaubt, sofern das Skript auf der lokalen Festplatte liegt. Entfernte Skripte können nur mit digitaler Signatur gestartet werden. Diese Einstellung nennt sich „RemoteSigned". In anderen Betriebssystemen gibt es aber keine Änderung des Standards, der „Restricted" lautet.

### 1.4.6 Skriptausführungsrichtlinie ändern

Um ein PowerShell-Skript auf Betriebssystemen wie Windows 7, Windows 8.x und Windows 10, wo dies im Standard nicht erlaubt ist, überhaupt starten zu können, müssen Sie die Skript-Ausführungsrichtlinie ändern. Später in diesem Buch lernen Sie, welche Optionen es dafür gibt. Für den ersten Test wird die Sicherheit ein wenig abgeschwächt, aber wirklich nur ein wenig. Mit dem folgenden Befehl lässt man die Ausführung von Skripten zu, die sich auf dem lokalen System befinden, verbietet aber Skripten von Netzwerkressourcen (das Internet eingeschlossen) die Ausführung, wenn diese keine digitale Signatur besitzen.

```
Set-ExecutionPolicy RemoteSigned
```

Später in diesem Buch lernen Sie, wie Sie PowerShell-Skripte digital signieren. Außerdem erfahren Sie, wie Sie Ihr System auf Skripte beschränken, die Sie oder Ihre Kollegen signiert haben.

Überprüfen Sie die vorgenommenen Änderungen mit dem Commandlet `Get-ExecutionPolicy`.

Es kann nun sein, dass Sie `Set-ExecutionPolicy` gar nicht ausführen können und eine Fehlermeldung wie die nachstehende sehen, dass die Änderung in der Registrierungsdatenbank mangels Rechten nicht ausgeführt werden konnte.

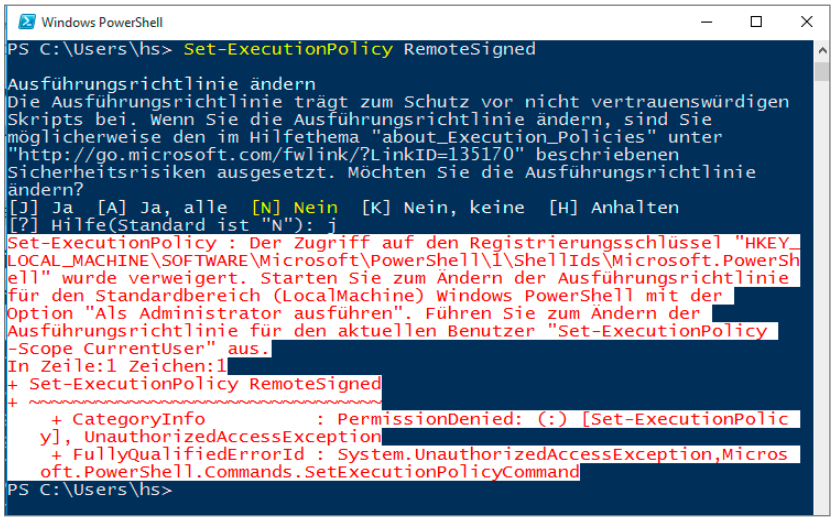



**Bild 1.17** Die Benutzerkontensteuerung verbietet die Änderung der Skriptausführungsrichtlinie

Dies ist die Benutzerkontensteuerung, die Microsoft seit Windows Vista in Windows mitliefert. Benutzerkontensteuerung (User Account Control, UAC) bedeutet, dass alle Anwendungen seit Windows Vista immer unter normalen Benutzerrechten laufen, auch wenn ein Administrator angemeldet ist. Wenn eine Anwendung höhere Rechte benötigt (z. B. administrative Aktionen, die zu Veränderungen am System führen), fragt Windows explizit in Form eines sogenannten Consent Interface beim Benutzer nach, ob der Anwendung diese Rechte gewährt werden sollen.

**HINWEIS:** Nur mit Windows Server ab Version 2012 startet der eingebaute Administrator (Konto „Administrator“) alle Skripte, die Konsole und andere .exe-Anwendungen unter vollen Rechten. Alle anderen Administratoren unterliegen der Benutzerkontensteuerung.

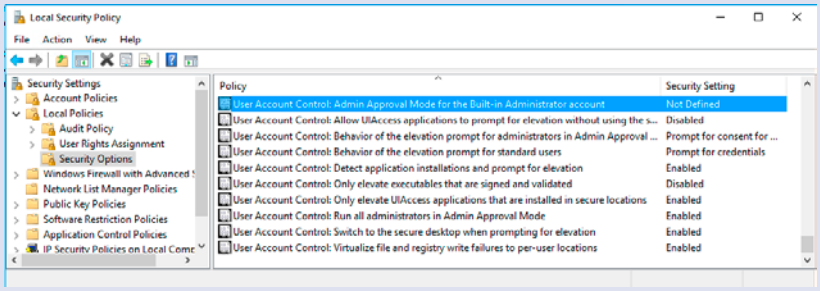



**Bild 1.18** Die besondere Einstellung für den eingebauten Administrator in den Sicherheitsrichtlinien von Windows Server

Um die PowerShell mit vollen Rechten zu starten, wählen Sie aus dem Startmenü (oder einer Verknüpfung z. B. in der Taskleiste) die PowerShell mit der rechten Maustaste aus und klicken auf „Als Administrator ausführen“.

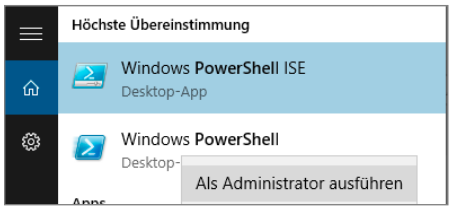



**Bild 1.19**
PowerShell „Als Administrator ausführen“

Dass die PowerShell als Administrator gestartet ist, sehen Sie an dem Zusatz „Administrator:“ in der Fenstertitelzeile der Konsole.

Geben Sie in diesem Fenster erneut ein:

```
Set-ExecutionPolicy RemoteSigned
```

Dies sollte nun funktionieren wie in der nachstehenden Bildschirmabbildung gezeigt.

Starten Sie nun das Skript erneut mit:

```
x:\temp\createuser.ps1
```

Jetzt sollte die Nachricht erscheinen, dass das Benutzerkonto erstellt worden ist.

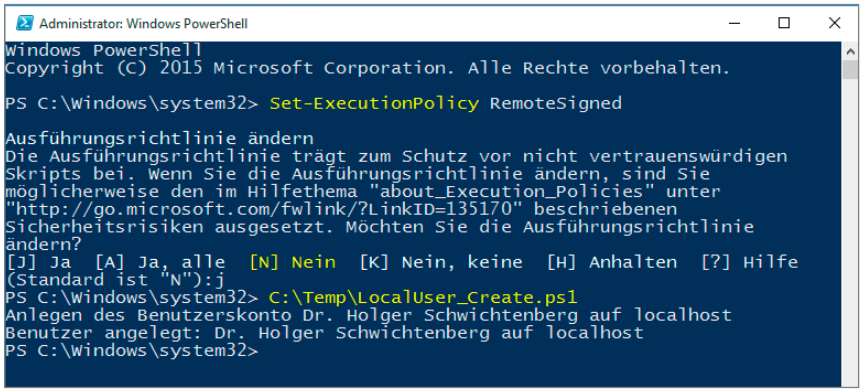



**Bild 1.20** Erfolgreiches Ändern der Skriptausführungsrichtlinien und Start des Skripts „LocalUser_Create.ps1“

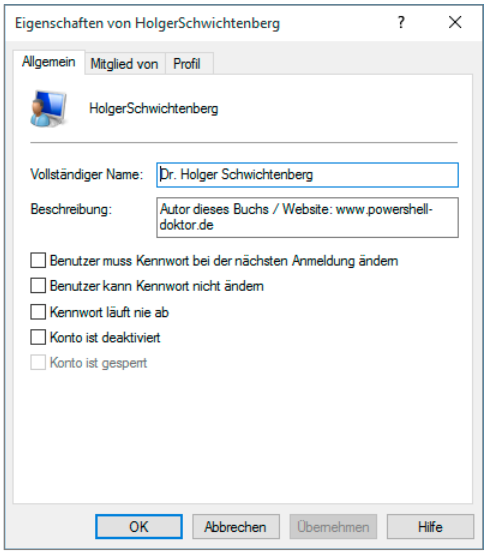



**Bild 1.21**
Das neu erstellte lokale Benutzerkonto

## 1.4.7 Farben ändern

Die PowerShell verwendet leider einige Farben mit wenig Kontrast. So werden Zeichenketten in einfachen oder doppelten Anführungszeichen in „DarkCyan“ auf dunkelblauem Grund dargestellt. Falls Sie dies nicht gut lesen können, ändern Sie doch die Farbe auf Cyan:

```
Set-PSReadlineOption -TokenKind String -ForegroundColor Cyan
```

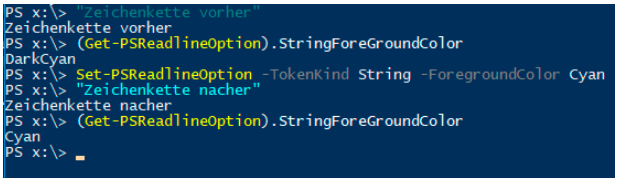



**Bild 1.22**
Auswirkung der Farbänderung

Falls Sie beim Eingeben schon einen Fehler gemacht haben, haben Sie rote Schrift auf blauem Untergrund gesehen. Wenn Sie das nicht gut lesen können, geben Sie bitte ein:

```
(Get-Host).PrivateData.ErrorBackgroundColor = "white"
```

Damit stellen Sie um auf rote Schrift auf weißem Grund für Fehlerausgaben.

So einen Befehl legt man in der PowerShell-Profilskriptdatei ab, damit er immer beim Start der PowerShell automatisch ausgeführt wird, siehe Buchteil B, Kapitel „Standardeinstellungen ändern mit Profilskripten“.

**HINWEIS:** Microsoft liefert Set-PSReadlineOption nur auf aktuellen Windows-Versionen mit. Auf älteren Windows-Versionen müssen Sie das Modul „PSReadline“ zunächst mit folgendem Befehl aus dem Netz herunterladen:

```
Install-Module psreadline
```

Dabei kommt es oft zu mehreren Sicherheitsabfragen, weil erst als Grundlage „NuGet“ auf Ihrem System installiert oder aktualisiert werden muss.

## 1.5 PowerShell Core installieren und testen

Dieses Unterkapitel behandelt die Installation der plattformneutralen PowerShell Core 6.x auf Windows, Linux und MacOS. Die PowerShell Core 6.x wird bislang mit keinem Betriebssystem direkt ausgeliefert.

Sie können dieses Unterkapitel überspringen, wenn Sie nur die klassische Windows PowerShell einsetzen wollen. Details zur PowerShell Core lesen Sie im Kapitel „PowerShell Core 6.x für Windows, Linux und MacOS“.

### 1.5.1 Installation und Test auf Windows

Für Windows wird die PowerShell Core 6.x sowohl als Installationsprogramm (MSI) als auch als ZIP-Archiv ausgeliefert [https://github.com/PowerShell/PowerShell/releases]. Die zum Redaktionsschluss aktuelle Version 6.0.1 enthält gegenüber Version 6.0 nur die Behebung von Sicherheitslücken.

**TIPP:** Die PowerShell Core 6.x kann man parallel zu den bisherigen Windows PowerShell-Installationen auf einem Windows-Rechner betreiben. Mit allen bisherigen PowerShell-Aktualisierungen war so ein Parallelbetrieb nicht möglich.

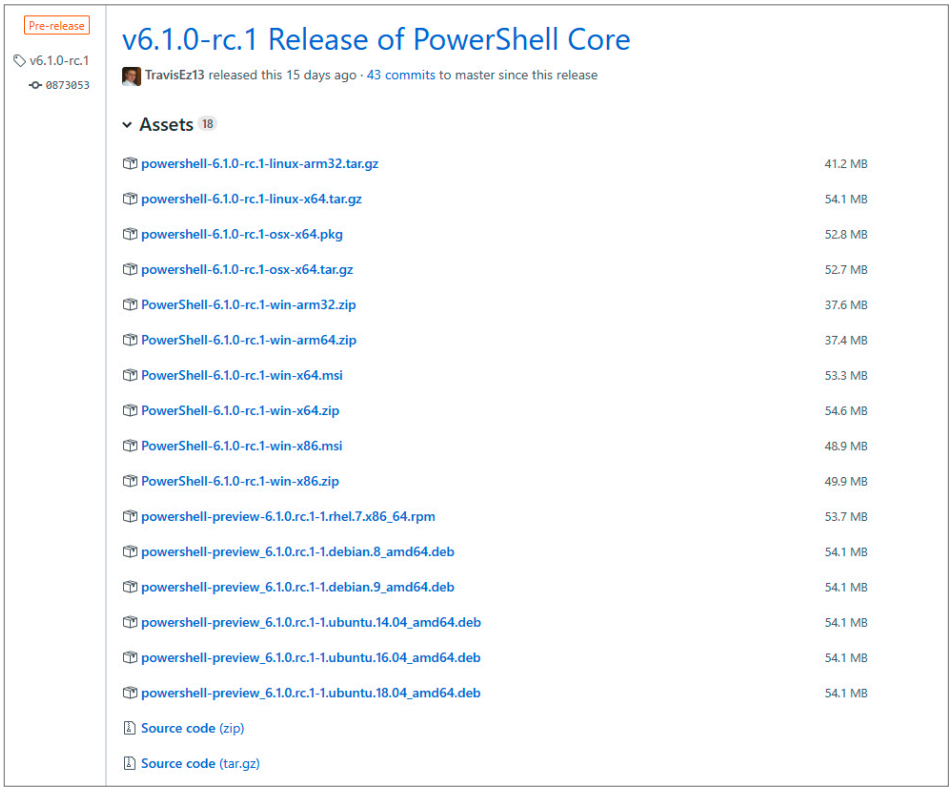



**Bild 1.23** Release-Übersicht von PowerShell Core 6.1

### 1.5.1.1 Installation mit der Archivdatei

Das Archiv enthält auch die benötigten Dateien von .NET Core 2.0. Man entpackt das Archiv und startet dann dort einfach vom Windows Explorer oder von der klassischen Kommandozeile oder der klassischen PowerShell aus die ausführbare Datei pwsh.exe.

**ACHTUNG:** In der Windows PowerShell war der Name der Programmdatei powershell.exe. Microsoft hat den Namen gegenüber Windows PowerShell 5.1 bewusst geändert, um den Parallelbetrieb einfacher zu machen. Auf einem Windows mit Windows PowerShell 5.1 und PowerShell Core 6.x startet man also per Eingabe von powershell.exe immer die Windows PowerShell und durch Eingabe von pwsh.exe immer die PowerShell Core.

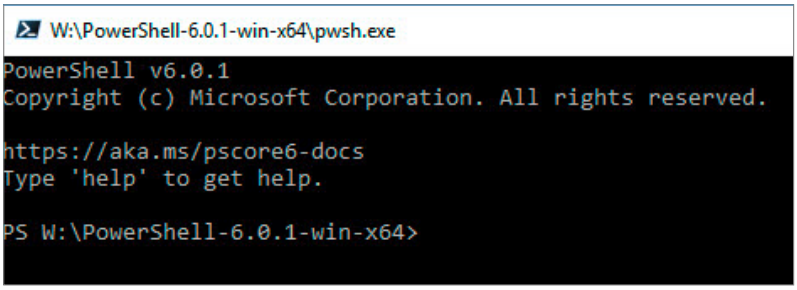



**Bild 1.24** So meldet sich die PowerShell Core 6.0.1 auf Windows.

### 1.5.1.2 MSI-basierte Installation

Das MSI-basierte Installationsprogramm installiert die PowerShell Core im Standard im Pfad c:\Program Files\PowerShell\. Dieser Pfad lässt sich bei der Installation ändern. In diesem Pfad wird ein Unterordner für die Version erstellt, z. B. 6.0.1. In diesem Ordner befinden sich dann alle Dateien der PowerShell Core inklusive der benötigten Dateien von .NET Core.

**TIPP:** Die MSI-basierte Installation hat den kleinen Vorteil, dass der Zielpfad der Installation automatisch zur Umgebungsvariablen %Path% ergänzt wird, sodass man nun pwsh.exe ohne Voranstellen eines Pfadnamens starten kann. Außerdem entsteht ein Eintrag im Startmenü (mit schwarzem statt blauem Symbol zur Unterscheidung von der Windows PowerShell).

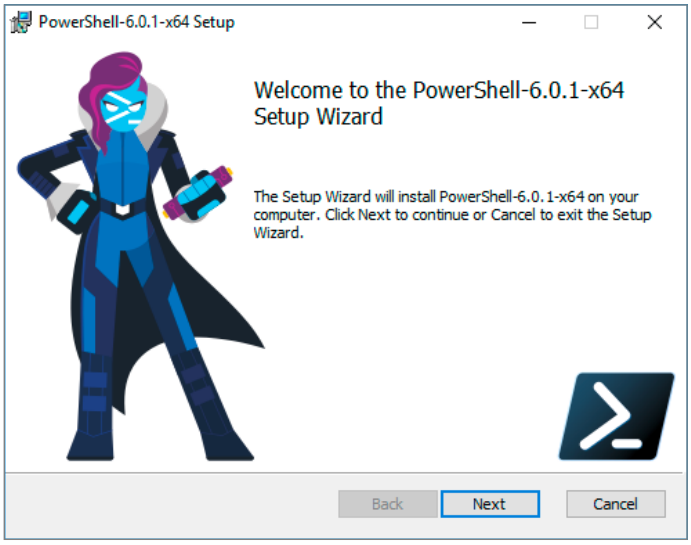



**Bild 1.25** Installation der PowerShell Core mit dem MSI-Paket

### 1.5.2 Installation und Test auf Ubuntu Linux

PowerShell Core 6.x wird für Ubuntu als .deb-Datei ausgeliefert [*https://github.com/PowerShell/PowerShell/releases*], die sich über das „Ubuntu Software Center“ (Ubuntu 14.04) bzw. „Ubuntu Software“ (Ubuntu ab Version 16.04) installieren lässt. Alternativ geht dies nach einem manuelle Download per Kommandozeile:

```
sudo apt install /home/hs/Downloads/powershell_6.0.1-1.ubuntu.16.04_amd64.
deb
sudo apt-get install -f
```

**HINWEIS:** Lassen Sie sich nicht verunsichern, wenn Warnungen bezüglich Abhängigkeiten beim ersten Befehl erscheinen. Diese Probleme werden durch den zweiten Befehl geheilt.

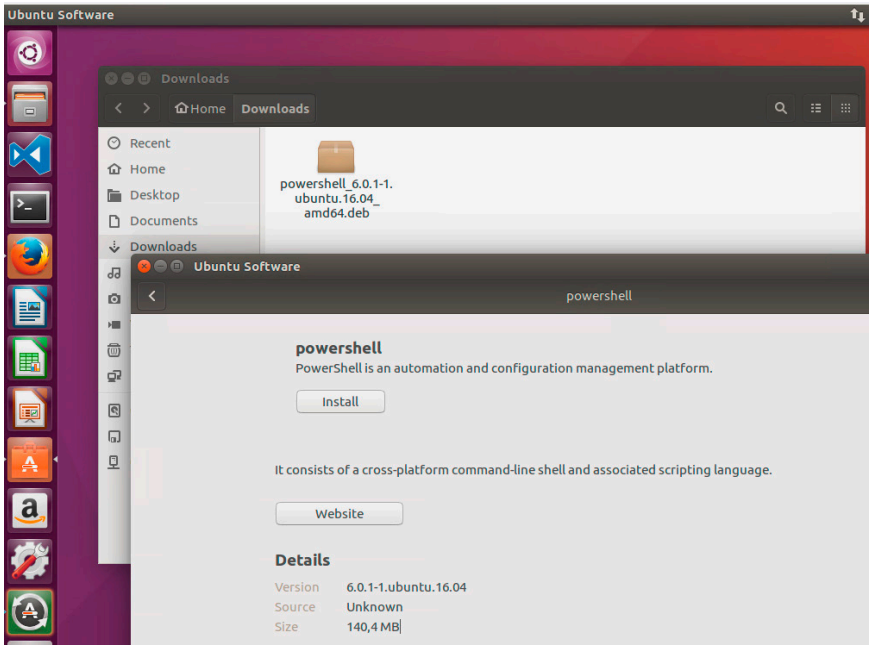



**Bild 1.26** Installation der PowerShell Core 6.x auf Ubuntu Linux

Für Debian und Kali gibt es ebenfalls eine .deb-Datei. Red Hat Enterprise Linux, OpenSUSE, und CentOS werden durch .rpm-Dateien unterstützt. Für andere Linux-Distributionen gibt es eine Archiv-Datei (.gz).

Zum Start der PowerShell Core gibt man im Terminal-Fenster `pwsh` (nicht powershell oder powershell.exe!) ein.

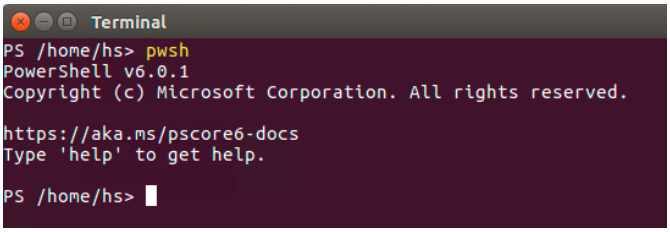



**Bild 1.27**
Start der PowerShell Core 6.0.1 auf Ubuntu-Linux

### 1.5.3 Installation und Test auf MacOS

Für die Installation auf MacOS stellt Microsoft unter [https://github.com/PowerShell/PowerShell/releases] eine .pkg-Datei (Apple Software Package) oder alternativ ein Archiv (.gz) bereit. Das Paket braucht auf dem aktuellen Stand rund 140 MB Festplattenplatz.

**HINWEIS:** Der Autor dieses Buchs besitzt kein eigenes MacOS-System. Alle Tests wurden auf einem in der Cloud gemieteten MacOS-System des Anbieters „macincloud“ (*www.macincloud.com*) durchgeführt.

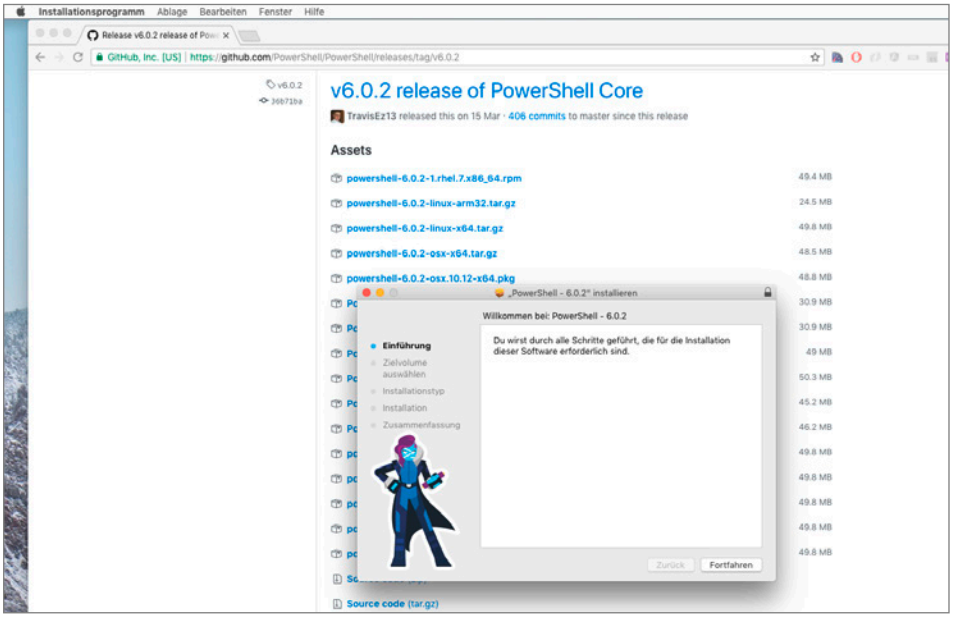



**Bild 1.28** Download und Start der .pgk-Datei

Zum Start der PowerShell Core auf OS/X gibt man im bash-basierten Terminal-Fenster `powershell` (nicht powershell.exe!) ein.

```
Last login: Thu Apr 26 16:21:30 on ttys002
Andres-MBP-5:~ andrekramer$ pwsh
PowerShell v6.0.2
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/pscore6-docs
Type 'help' to get help.

PS /Users/andrekramer>
```

**Bild 1.29** Start der PowerShell Core 6.x auf MacOS

**TIPP:** Microsoft verwendet auch unter MacOS verschiedene Farben an der Konsole, die aber in einigen Fällen (z. B. Commandlet-Namen und Klassenmitgliedernamen) hell sind und auf einem weißen Hintergrund nicht genug Kontrast bieten. Stellen Sie daher für das MacOS-Terminal-Fenster ein Farbschema mit einem dunkleren Hintergrund ein. Gut eignet sich das Farbschema „Ocean“. Sie ändern das Farbschema in dem Terminal-Fenster im Menü „Shell/Show Inspector“ unter „Settings“.

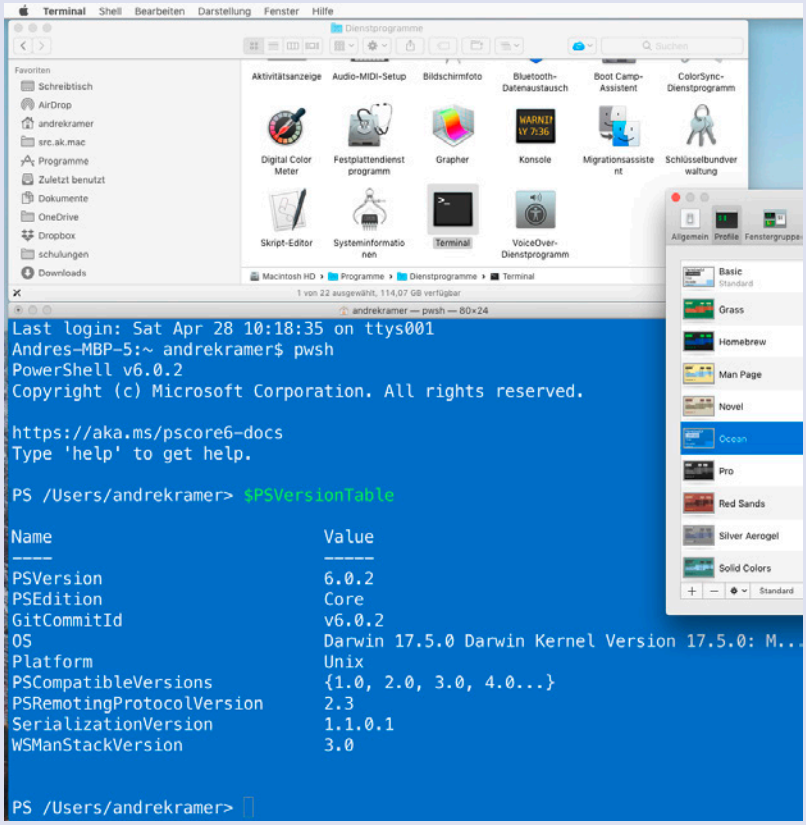



**Bild 1.30** Einstellung des Farbschemas „Ocean“ für das MacOS-Terminal-Fenster

## 1.6 Woher kommen die Commandlets?

Die Windows PowerShell umfasste in der Version 1.0 nur 129 Commandlets (und Funktionen). In PowerShell 2.0 waren es 236, in PowerShell 3.0 waren es 322 und in PowerShell 4.0 sind es auch immer noch „nur“ 328 und in PowerShell 5.0 unter Windows 10 sind es 340, in Windows 10 Creators Update (Redstone 2, Version 1703 vom April 2017) mit PowerShell 5.1 sind es 370. Als Kern der PowerShell werden hier alle Commandlets und Funktionen bezeichnet, die sich in einem der PowerShell-Module befinden, die mit Windows ausgeliefert werden bzw. mit dem WMF-Add-on installiert werden und die auf allen unterstützten Betriebssystemen verfügbar sind (und daher das Wort „PowerShell“ im Modulnamen tragen und in der Dokumentation „Core Modules“ genannt werden).

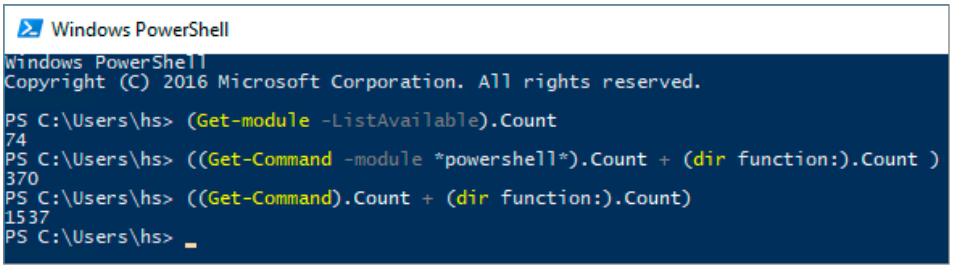



**Bild 1.31** Zählen der Commandlets und Funktionen unter Windows 10 (Stand Creators Update, Versionsnummer 1703, Redstone 2)

Es gibt noch viel mehr Commandlets als die oben genannten, diese gehören aber nicht zur Windows PowerShell im engeren Sinne, sondern zu optionalen Erweiterungen oder der jeweiligen Windows-Betriebssystemversion.

Schon kurz nach Version 1.0 der Windows PowerShell gab es erste Erweiterungen wie zum Beispiel die PowerShell Community Extensions (siehe nächstes Unterkapitel).

Mit Windows 7 bzw. Windows Server 2008 R2 hat Microsoft begonnen, Zusatzmodule direkt mit dem Betriebssystem auszuliefern. Diese Zusatzmodule bringen in Windows 8.1 die Anzahl der Commandlets auf über 1000. In Windows 10 (Stand Creators Update, Versionsnummer 1703) sind es dann 1537.

**ACHTUNG:** Anders als die Erweiterungsmodule, die es oft für mehrere (auch ältere) PowerShell-Versionen gibt, kann man die zum Betriebssystem gehörenden Module nicht in einem älteren Betriebssystem verwenden. In dem zum Redaktionsschluss dieses Buchs Stand der PowerShell 6.0.1 kann man viele, aber nicht noch nicht alle zum Windows-Betriebssystem gehörenden PowerShell-Module auch in PowerShell Core unter Windows verwenden.

## 1.7 PowerShell Community Extensions (PSCX) herunterladen und installieren

Bei den „PowerShell Community Extensions“ (kurz PSCX) handelt es sich um ein Open Source-Projekt (ursprünglich auf Codeplex.com, mittlerweile auf Github.com, siehe [https://github.com/Pscx/Pscx], das zusätzliche Funktionalität mit Commandlets für die Windows PowerShell realisiert, wie zum Beispiel `Get-DHCPServer`, `Get-DomainController`, `Get-MountPoint`, `Get-TerminalSession`, `Set-VolumneLabel`, `Write-Tar` und viele weitere. Das Projekt steht unter Führung von Microsoft, aber jeder .NET-Softwareentwickler ist eingeladen, daran mitzuwirken. In regelmäßigen Abständen werden neue Versionen veröffentlicht. Die aktuelle Version zum Reaktionsschluss dieses Buchs ist die Version 3.3.2.

**TIPP:** In diesem Buch werden an einigen Stellen Commandlets aus den PSCX verwendet. Daher sollten Sie die PSCX installieren.

Die Installation der PSCX führt man heutzutage über das Commandlet `Install-Module` aus. Dieses Commandlet lädt das Modul aus der PowerShell Gallery [https://www.powershellgallery.com], einem von Microsoft betriebenen Online-Portal mit PowerShell-Erweiterungen und installiert das Module. Alternativ dazu können Sie auf Github ein ZIP-Paket laden [https://github.com/Pscx/Pscx/releases] und die Installation manuell vornehmen.

Für die automatische Installation führen Sie in einer PowerShell-Konsole, die administrative Rechte besitzt, bitte aus:

```
Install-Module Pscx -Scope CurrentUser -AllowClobber
```

Die nächste Bildschirmabbildung erklärt, die Bedeutung des Parameters `-AllowClobber`: In den PSCX gibt es einige Befehle, die mittlerweile in die PowerShell festeingebaut sind.

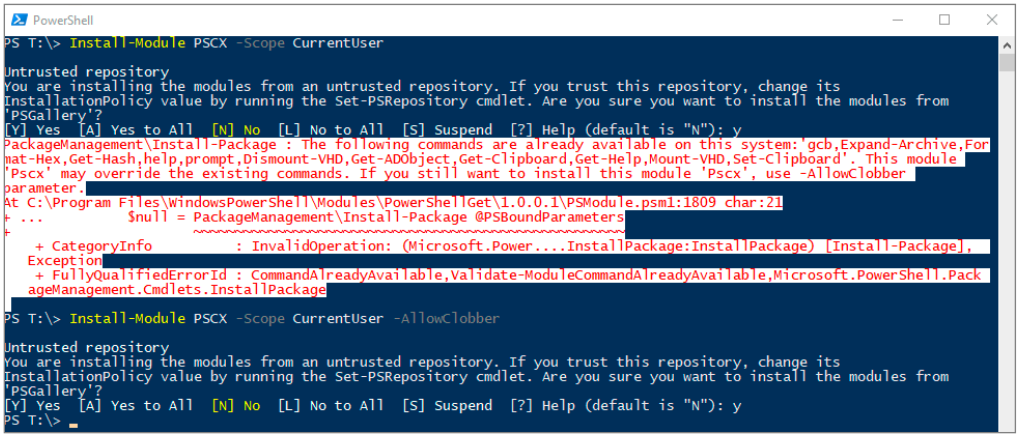



**Bild 1.32** Installation der PSCX

Geben Sie nun `Get-DomainController` ein (wenn Ihr Computer Mitglied eines Active Directory ist) oder testen Sie die PSCX mit dem Befehl `Ping-Host`, der auf jedem Computer im Netzwerk funktioniert. Wie Sie in der Bildschirmabbildung an der Ausgabe zu `Ping-Host` lesen können: Es ist ein Commandlet für das es mittlerweile in der PowerShell einen Ersatz (hier: `Test-Connection`) gibt.

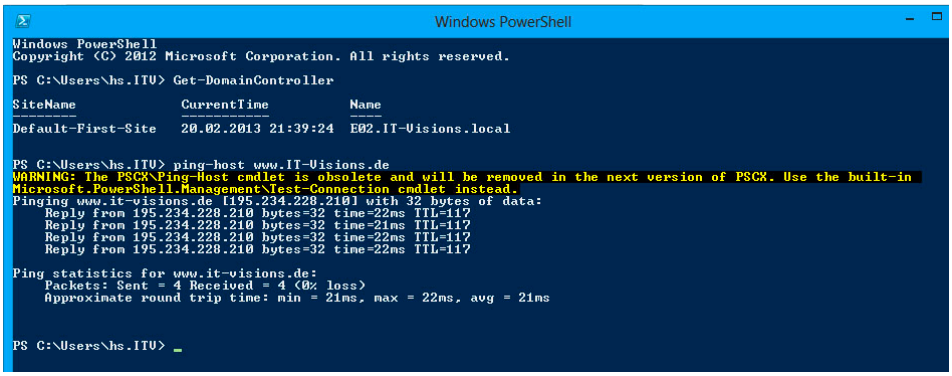



**Bild 1.33** PSCX-Befehle Get-DomainController und Ping-Host testen

## 1.8 Den Windows PowerShell-Editor „ISE“ verwenden

Integrated Scripting Environment (ISE) ist der Name des Skripteditors, den Microsoft seit der Windows PowerShell 2.0 mitliefert und der in Windows PowerShell 3.0 nochmals erheblich verbessert wurde. Die ISE startet man mit dem Symbol „PowerShell ISE“ oder indem man in der PowerShell den Befehl „ise“ ausführt.

Die ISE verfügt über zwei Fenster: ein Skriptfenster (im Standard oben, alternativ über „View“-Menü einstellbar rechts) und ein interaktives Befehlseingabefenster (unten bzw. links). Optional kann man ein drittes Fenster einblenden, das „Command Add-On“, in dem man Befehle suchen kann und eine Eingabehilfe für Befehlsparameter erhält.

Geben Sie unten im interaktiven Befehlseingabefenster in der ISE ein:

```
Get-Process
```

Nachdem Sie mindestens einen Buchstaben eingegeben haben, können Sie die Eingabe mit der Tabulatortaste vervollständigen. Alternativ können Sie **STRG+Leertaste** drücken für eine Eingabehilfe mit Auswahlfenster (IntelliSense). Die Ausgaben des interaktiven Bereichs erscheinen dann direkt unter den Befehlen, wie bei der PowerShell-Konsole. Einen dedizierten Ausgabebereich wie in der ISE in PowerShell 2.0 gibt es nicht mehr.

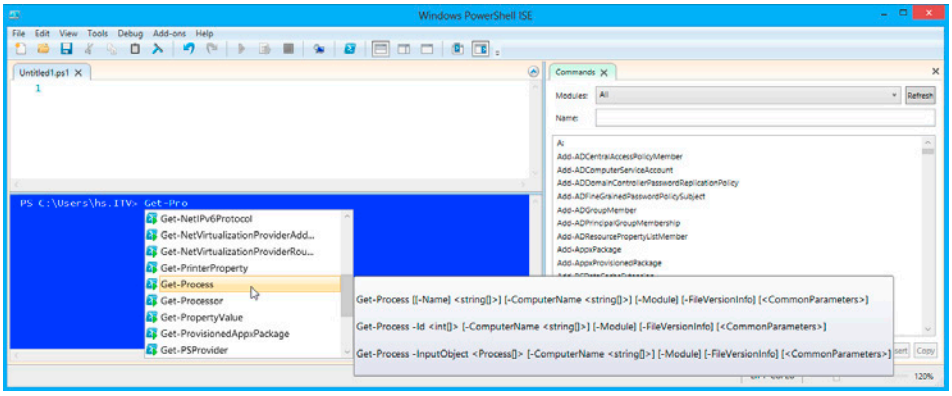



**Bild 1.34** IntelliSense-Eingabehilfe

Um die ISE im Skriptmodus zu verwenden, erstellen Sie eine neue Skriptdatei (Menü „File/New“) oder öffnen Sie eine vorhandene *.ps1*-Datei (Menü „File/Open“). Öffnen Sie als Beispiel die Skriptdatei *CreateUser.ps1*, die Sie zuvor erstellt haben. Es sind Zeilennummern zu sehen. Die verschiedenen Bestandteile des Skripts sind in unterschiedlichen Farben dargestellt. Auch hier funktioniert die Eingabeunterstützung mit der Tabulatortaste und IntelliSense.

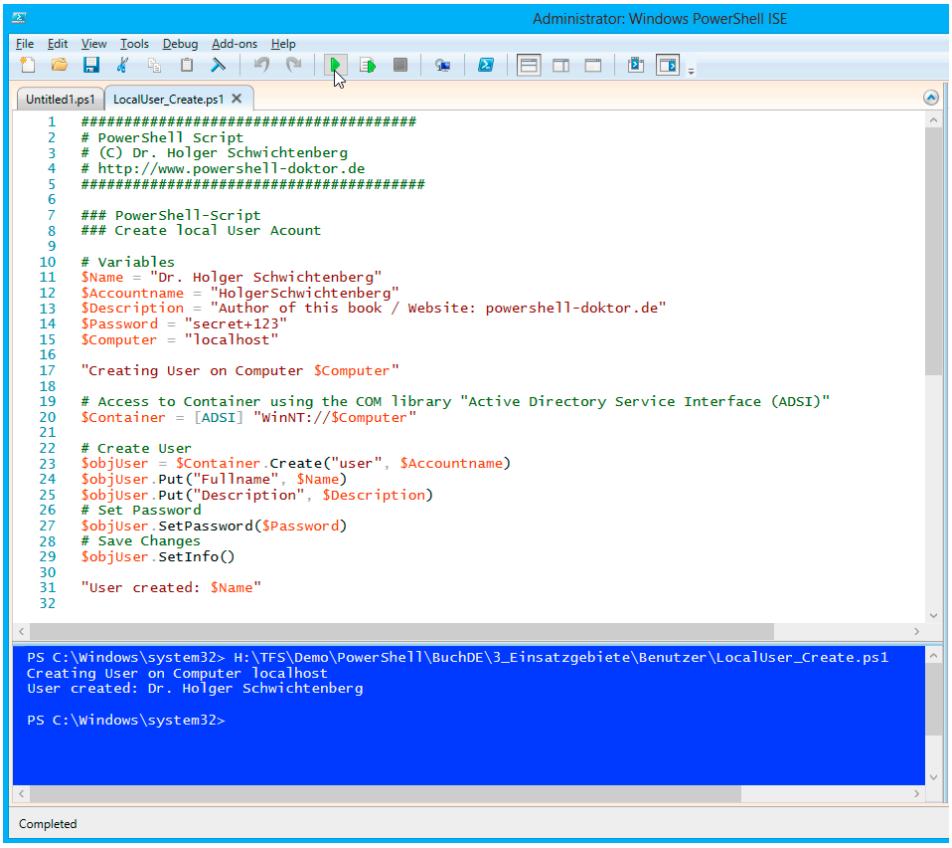



**Bild 1.35** Die ISE im Skriptmodus

Um das Skript auszuführen, klicken Sie auf das Start-Symbol in der Symbolleiste (siehe die Screenshots) oder drücken Sie **F5**. Auch hier wird das Ergebnis im interaktiven Bereich angezeigt.

**TIPP:** Stellen Sie sicher, dass Sie die ISE als Administrator ausführen und dass das Benutzerkonto noch nicht existiert, bevor Sie das Skript ausführen.

Ein interessantes Feature ist das Debugging, mit dem Sie ein Skript Zeile für Zeile durchlaufen und währenddessen den Zustand der Variablen betrachten können.

Setzen Sie dazu den Cursor auf eine beliebige Zeile in Ihrem Skript und tippen Sie dann auf **F9** (oder wählen Sie „Toogle Breakpoint“ im Kontextmenü oder im Menü „Debug“). Daraufhin erscheint die Zeile in Rot – ein sogenannter „Haltepunkt“.

Starten Sie das Skript nun mit **F5**. Die ISE stoppt in der Zeile mit dem Haltepunkt und diese wird orange. Mit der Taste **F10** springen Sie zum nächsten Befehl. Diese wird dann gelb und die Zeile mit dem Haltepunkt wird wieder rot.

**HINWEIS:** Die gelbe Zeile ist immer die nächste Zeile, die ausgeführt wird.

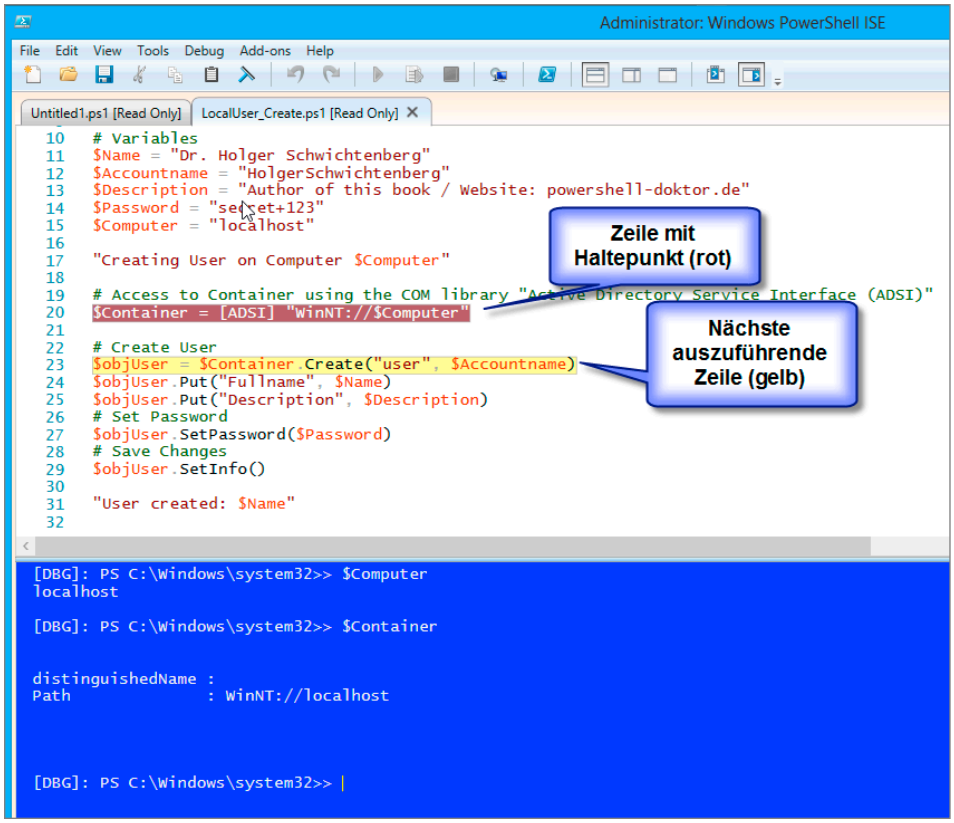



**Bild 1.36** Skript-Debugging mit der ISE

Im interaktiven Bereich können Sie im Haltemodus den aktuellen Zustand der Variablen abfragen, indem Sie dort z. B. eingeben

```
$Computer
```

oder

```
$Container
```

Man kann auch Werte interaktiv ändern. Um das Skript fortzusetzen, drücken Sie wieder **F5**. Über das Menü „Debug“ sind weitere Steuerbefehle möglich.

**HINWEIS:** Sie müssen den Debugger vorher beenden (Menüpunkt „Debug/ Stop Debugger“), wenn Sie das Skript erneut ändern möchten.

# 2 Fakten zur PowerShell

## 2.1 Geschichte der PowerShell

Das Active Scripting ist einigen Administratoren zu komplex, weil es viel Wissen über objektorientiertes Programmieren und das Component Object Model (COM) voraussetzt. Die vielen Ausnahmen und Ungereimtheiten im Active Scripting erschweren das Erlernen von Windows Script Host (WSH) und der zugehörigen Komponentenbibliotheken.

Schon im Zuge der Entwicklung des Windows Server 2003 gab Microsoft zu, dass man Unix-Administratoren zum Interview über ihr tägliches Handwerkszeug gebeten hatte. Das kurzfristige Ergebnis war eine große Menge zusätzlicher Kommandozeilenwerkzeuge. Langfristig setzt Microsoft jedoch auf eine Ablösung des DOS-ähnlichen Konsolenfensters durch eine neue Skripting-Umgebung.

Mit dem Erscheinen des .NET Frameworks im Jahre 2002 wurde lange über einen WSH.NET spekuliert. Microsoft stellte jedoch die Neuentwicklung des WSH für das .NET Framework ein, als abzusehen war, dass die Verwendung von .NET-basierten Programmiersprachen wie C# und Visual Basic .NET dem Administrator nur noch mehr Kenntnisse über objektorientierte Softwareentwicklung abverlangen würde.

Microsoft beobachtete in der Unix-Welt eine hohe Zufriedenheit mit den dortigen Kommandozeilen-Shells und entschloss sich daher, das Konzept der Unix-Shells, insbesondere das Pipelining, mit dem .NET Framework zusammenzubringen und daraus eine .NET-basierte Windows Shell zu entwickeln. Diese sollte noch einfacher als eine Unix-Shell, aber dennoch so mächtig wie das .NET Framework sein.

In einer ersten Beta-Version wurde die neue Shell schon unter dem Codenamen „Monad“ auf der Professional Developer Conference (PDC) im Oktober 2003 in Los Angeles vorgestellt. Nach den Zwischenstufen „Microsoft Shell (MSH)“ und „Microsoft Command Shell“ trägt die neue Skriptumgebung seit Mai 2006 den Namen „Windows PowerShell“.

Die PowerShell 1.0 erschien am 6.11.2006 zeitgleich mit Windows Vista, war aber dort nicht enthalten, sondern musste heruntergeladen und nachinstalliert werden.

Die PowerShell 2.0 ist zusammen mit Windows 7/Windows Server 2008 R2 erschienen am 22.7.2009.

Die PowerShell 3.0 ist zusammen mit Windows 8/Windows Server 2012 erschienen am 15.8.2012.

Die PowerShell 4.0 ist zusammen mit Windows 8.1/Windows Server 2012 R2 am 9.9.2013 erschienen.

Die PowerShell 5.0 ist als Teil von Windows 10 erschienen am 29.7.2015. Abweichend von den bisherigen Gepflogenheiten ist die PowerShell 5.0 als Erweiterung für Windows Server 2008 R2 (mit Service Pack 1) und Windows Server 2012/2012 R2 erst deutlich später am 16.12.2015 erschienen. Für Windows 7 und Windows 8.1 sollte es erst gar keine Version mehr geben. Doch am 18.12.2015 hatte Microsoft ein Einsehen mit den Kunden und lieferte die PowerShell 5.0 auch für diese Betriebssysteme nach. Kurioserweise musste Microsoft den Download dann am 23.12.2015 wegen eines gravierenden Fehlers für einige Wochen vom Netz nehmen. Microsoft hatte das Produkt im neuen Agilitäts-Wahn nicht richtig getestet.

Der Windows Server 2016 (erschienen am 26.9.2016) enthält PowerShell 5.1. und Windows 10 und wurde mit dem Windows 10 Anniversary Update (Version 1607, Codename „Redstone 1“) am 2.8.2016 aktualisiert. PowerShell 5.1 ist erst seit 19.1.2017 als Add-on für Windows 7, Windows 8.1, Windows Server 2008 R2, Windows 2012 und Windows 2012 R2 verfügbar.

Eine reduzierte „Core“-Version der Windows PowerShell ist als „Windows PowerShell Core 5.1“ enthalten in Windows Nano Server, im ersten Release 2016 als Standardpaket, im zweiten in Release „1709“ als Option.

Die erste Version der plattformneutralen PowerShell Core (ohne Windows im Namen!) ist mit der Versionsnummer 6.0 am 20.01.2018 erschienen. Zum Redaktionsschluss der Aktualisierung dieses Buchs am 6.9.2018 ist die Version PowerShell Core 2.1 in Arbeit und als „Release Candidate“-Version verfügbar.

**HINWEIS:** Mit Windows 10 hat Microsoft das Auslieferungsverfahren auf „Windows as a Service“ umgestellt. Dies bedeutet, dass Microsoft über Windows Update im Sinne der neuen „agilen“ Strategie nun auch ständig neue Funktionen ausliefert. Dies betrifft ebenso die Windows PowerShell, die dann zukünftig auch auf diesem Wege häufigere Aktualisierungen erfahren kann. Wie häufig dies sein wird, ist zum Reaktionsschluss dieses Buchs noch offen.

Microsoft hat sich seit dem Jahr 2015 für andere Betriebssysteme und die Entwicklung als „Open Source Software“ (OSS) geöffnet. Dies betrifft nun auch die PowerShell: Die PowerShell Core, die am 20.1.2018 als Version 6.0 (in Nachfolge von Windows PowerShell 5.1) erschienen ist, ist Open Source läuft nicht nur auf Windows, sondern auch MacOS und Linux.

## 2.2 Motivation zur PowerShell

Falls Sie eine Motivation brauchen, sich mit der PowerShell zu beschäftigen, wird dieses Kapitel sie Ihnen liefern. Es stellt die Lösung für eine typische Scripting-Aufgabe sowohl im „alten" Windows Script Host (WSH) als auch in der „neuen" PowerShell vor.

Zur Motivation, sich mit der PowerShell zu beschäftigen, soll folgendes Beispiel aus der Praxis dienen. Es soll ein Inventarisierungsskript für Software erstellt werden, das die installierten MSI-Pakete mit Hilfe der Windows Management Instrumentation (WMI) von mehreren Computern ausliest und die Ergebnisse in einer CSV-Datei *(softwareinventar.csv)* zusammenfasst. Die Namen (oder IP-Adressen) der abzufragenden Computer sollen in einer Textdatei *(computernamen.txt)* stehen.

Die Lösung mit dem WSH benötigt 90 Codezeilen (inklusive Kommentare und Parametrisierungen). In der PowerShell lässt sich das Gleiche in nur 13 Zeilen ausdrücken. Wenn man auf die Kommentare und die Parametrisierung verzichtet, dann reicht sogar genau eine Zeile. Das PowerShell-Skript läuft in der Windows PowerShell und auch in der PowerShell Core unter Windows, aber nicht unter Linux und MacOS, da es dort noch keine Implementierung des für den Zugriff auf die installierte Software notwendigen Web Based Enterprise Management (WBEM) und des Common Information Model (CIM) für die PowerShell gibt.

**Listing 2.1** Softwareinventarisierung – Lösung 1 mit dem WSH
[3_Einsatzgebiete/Software/Software_Inventory.vbs]

```
' ----------------------------------------
' Skriptname: Software_inventar.vbs
' Autor: Dr. Holger Schwichtenberg
' ----------------------------------------
' Dieses Skript erstellt eine Liste
' der installierten Software
' ----------------------------------------
Option Explicit

' --- Einstellungen
Const Trennzeichen = ";" ' Trennzeichen für Spalten in der Ausgabedatei
Const Eingabedateiname = "computernamen.txt"
Const Ausgabedateiname = "softwareinventar.csv"
Const Bedingung = "SELECT * FROM Win32_Product where not Vendor like '%Microsoft%'"

Dim objFSO ' Dateisystem-Objekt
Dim objTX  ' Textdatei-Objekt für die Liste der zu durchsuchenden computer
Dim i  ' Zähler für Computer
Dim computer ' Name des aktuellen computers
Dim Eingabedatei' Name und Pfad der Eingabedatei
Dim Ausgabedatei' Name und Pfad der Ausgabedatei

' --- Startmeldung
WScript.Echo "Softwareinventar.vbs"
WScript.Echo "(C) Dr. Holger Schwichtenberg, http://www.Windows-Scripting.de"

' --- Global benötigtes Objekt
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

```
' --- Ermittlung der Pfade
Eingabedatei = GetCurrentPfad & "\" & Eingabedateiname
Ausgabedatei = GetCurrentPfad & "\" & Ausgabedateiname

' --- Auslesen der computerliste
Set objTX = objFSO.OpenTextFile(Eingabedatei)

' --- Meldungen
WScript.Echo "Eingabedatei: " & Eingabedatei
WScript.Echo "Ausgabedatei: " & Ausgabedatei

' --- Überschriften einfügen
Ausgabe _
"computer" & Trennzeichen & _
"Name" & Trennzeichen & _
    "Beschreibung" & Trennzeichen & _
    "Identifikationsnummer" & Trennzeichen & _
    "Installationsdatum" & Trennzeichen & _
    "Installationsverzeichnis" & Trennzeichen & _
    "Zustand der Installation" & Trennzeichen & _
    "Paketzwischenspeicher" & Trennzeichen & _
    "SKU Nummer" & Trennzeichen & _
    "Hersteller" & Trennzeichen & _
    "Version"

' --- Schleife über alle Computer
Do While Not objTX.AtEndOfStream
    computer = objTX.ReadLine
    i = i + 1
    WScript.Echo "=== Computer #" & i & ": " & computer

GetInventar computer

Loop

' --- Eingabedatei schließen
objTX.Close
' --- Abschlußmeldung
WScript.echo "Softwareinventarisierung beendet!"

' === Softwareliste für einen computer erstellen
Sub GetInventar(computer)

Dim objProduktMenge
Dim objProdukt
Dim objWMIDienst

' --- Zugriff auf WMI
Set objWMIDienst = GetObject("winmgmts:" &_
    "{impersonationLevel=impersonate}!\\" & computer &_
    "\root\cimv2")
' --- Liste anfordern
Set objProduktMenge = objWMIDienst.ExecQuery _
    (Bedingung)
' --- Liste ausgeben
WScript.Echo "Auf " & computer & " sind " & _
objProduktMenge.Count & " Produkte installiert."
For Each objProdukt In objProduktMenge
```

```
    Ausgabe _
    computer & Trennzeichen & _
    objProdukt.Name & Trennzeichen & _
    objProdukt.Description & Trennzeichen & _
    objProdukt.IdentifyingNumber & Trennzeichen & _
    objProdukt.InstallDate & Trennzeichen & _
    objProdukt.InstallLocation & Trennzeichen & _
    objProdukt.InstallState & Trennzeichen & _
    objProdukt.PackageCache & Trennzeichen & _
    objProdukt.SKUNumber & Trennzeichen & _
    objProdukt.Vendor & Trennzeichen & _
    objProdukt.Version
WScript.Echo     objProdukt.Name
Next
End Sub

' === Ausgabe
Sub Ausgabe(s)
Dim objTextFile
' Ausgabedatei öffnen
Set objTextFile = objFSO.OpenTextFile(Ausgabedatei, 8, True)
objTextFile.WriteLine s
objTextFile.Close
'WScript.Echo s
End Sub

' === Pfad ermitteln. in dem das Skript liegt
Function GetCurrentPfad
GetCurrentPfad = objFSO.GetFile (WScript.ScriptFullName).ParentFolder
End Function
```

**Listing 2.2** Softwareinventarisierung – Lösung 2 als PowerShell-Skript
[3_Einsatzgebiete/Software/SoftwareInventory_WMI_Script.ps1]

```
# Einstellungen
$InputFileName = "computernamen.txt"
$OutputFileName = "softwareinventar.csv"
$Query = "SELECT * FROM Win32_Product where not Vendor like '%Microsoft%'"

# Einbabedatei auslesen
$Computers = Get-Content $InputFileName

# Schleife über alle Computer
$Software = $Computers | foreach { Get-CimInstance -query $Query -computername $_ }
# Ausgabe in CSV
$Software | select Name, Description, IdentifyingNumber, InstallDate,
InstallLocation, InstallState, SKUNumber, Vendor, Version | export-csv
$OutputFileName -notypeinformation
```

**Listing 2.3** Softwareinventarisierung – Lösung 3 als PowerShell-Pipeline-Befehl
[3_Einsatzgebiete/Software/SoftwareInventory_WMI_Pipeline.ps1]

```
Get-Content "computers.txt" | foreach {Get-CimInstance -computername $_ -query
"SELECT * FROM Win32_Product where not Vendor like '%Microsoft%'" } | export-csv
"Softwareinventory.csv" -notypeinformation
```

## 2.3 Betriebssysteme mit vorinstallierter PowerShell

Die folgende Tabelle zeigt, in welchen Betriebssystemen welche Version der PowerShell mitgeliefert wird bzw. wo sie nachträglich installierbar ist.

**Tabelle 2.1** Verfügbarkeit der PowerShell auf verschiedenen Betriebssystemen

| Betriebssystem | Mitgelieferte PowerShell | Nachträglich installierbare PowerShell |
|---|---|---|
| Windows 2000, Windows 9x, Windows ME, Windows NT 4.0 | PowerShell nicht enthalten | Nachträgliche Installation nicht von Microsoft unterstützt |
| Windows XP | PowerShell nicht enthalten | PowerShell 1.0 und PowerShell 2.0 |
| Windows Server 2003 | PowerShell nicht enthalten | PowerShell 1.0 und PowerShell 2.0 |
| Windows Server 2003 R2 | PowerShell nicht enthalten | PowerShell 1.0 und PowerShell 2.0 |
| Windows Vista | PowerShell nicht enthalten | PowerShell 1.0 und PowerShell 2.0 |
| Windows Vista | PowerShell 1.0 enthalten | PowerShell 2.0 |
| Windows Server 2008 | PowerShell 1.0 enthalten als optionales Features | PowerShell 2.0, PowerShell 3.0 |
| Windows Server 2008 R2 | PowerShell 1.0 enthalten | PowerShell 2.0, PowerShell 3.0 |
| Windows 7 | PowerShell 2.0 enthalten | PowerShell 3.0, PowerShell 4.0, PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x |
| Windows Server 2008 R2 | PowerShell 2.0 enthalten | PowerShell 3.0, PowerShell 4.0, PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x |
| Windows Server 2008 Core | PowerShell nicht enthalten | PowerShell 3.0, PowerShell Core 6.x |
| Windows Server 2008 R2 Core | PowerShell 2.0 enthalten als optionales Feature | PowerShell Core 6.x |
| Windows 8.0 | PowerShell 3.0 enthalten | Achtung: PowerShell 4.0 und 5.0/5.1 können nur durch ein (vorheriges) Update auf Windows 8.1 nachgerüstet werden. |

| Betriebssystem | Mitgelieferte PowerShell | Nachträglich installierbare PowerShell |
|---|---|---|
| Windows Server 2012 inkl. Core | PowerShell 3.0 enthalten | PowerShell 4.0, PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x |
| Windows 8.1 | PowerShell 4.0 enthalten | PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x |
| Windows Server 2012 R2 inkl. Core | PowerShell 4.0 enthalten | PowerShell 5.0, PowerShell 5.1, PowerShell Core 6.x |
| Windows 10 | PowerShell 5.0 enthalten | PowerShell Core 6.x |
| Windows 10 Creators Update (Redstone 2, Version 1703, April 2017) | PowerShell 5.1 enthalten | PowerShell Core 6.x |
| Windows Server 2016 | PowerShell 5.1 enthalten | PowerShell Core 6.x |
| Windows Server 1709 | PowerShell 5.1 enthalten | PowerShell Core 6.x |
| Windows Nano Server 2016 | Reduzierte PowerShell Core 5.1 enthalten | PowerShell Core 6.x |
| Windows Nano Server 1709 | – (PowerShell Core wurde aus dem Standardinstallationsumfang entfernt, vgl. [https://docs.microsoft.com/de-de/windows-server/get-started/nano-in-semi-annual-channel]) | PowerShell Core 5.1, PowerShell Core 6.x |
| Suse-Linux ab Version 42.1 | – | PowerShell Core 6.x |
| Ubuntu-Linux ab Version 14.04 | – | PowerShell Core 6.x |
| MacOS/X ab Version 10.12 | – | PowerShell Core 6.x |

## 2.4 Einflussfaktoren auf die Entwicklung der PowerShell

Die Windows PowerShell ist eine Symbiose aus:

- dem DOS-Kommandozeilenfenster,
- den bekannten Skript- und Shell-Sprachen wie Perl, Ruby, ksh und bash,
- dem .NET Framework und
- der Windows Management Instrumentation (WMI).

Die PowerShell ist implementiert auf dem .NET Framework. Sie ist jedoch kein .NET Runtime Host mit der Möglichkeit, Befehle der Common Intermediate Language (CIL) auf der Common Language Runtime (CLR) auszuführen.

Die PowerShell verwendet ein völlig anderes Host-Konzept mit Commandlets, Objekt-Pipelines und einer neuen Sprache, die von Microsoft als PowerShell Language (PSL) bezeichnet wird. Sie ist Perl, Ruby, C# und einigen Unix-Shell-Sprachen sehr ähnlich, aber mit keiner Unix-Shell kompatibel. Nutzer der WMI Command Shell *(wmic.exe)*, die mit Windows XP eingeführt wurde, werden sich in der PowerShell schnell zurechtfinden.

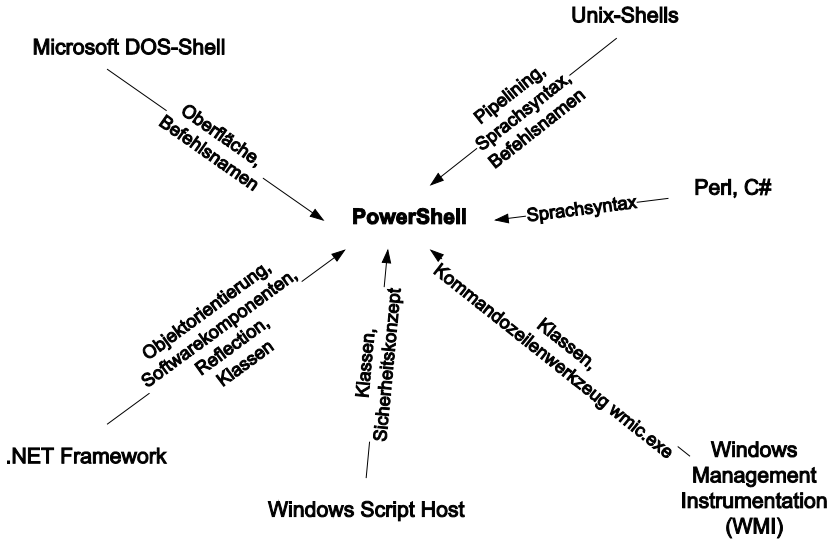



**Bild 2.1** Einflussfaktoren auf die Architektur und die Umsetzung der PowerShell

**ACHTUNG:** Die PowerShell ist angetreten, vom Administrator weniger Kenntnisse in Objektorientierung und über Softwarekomponenten zu verlangen, als dies der Vorgänger Windows Script Host (WSH) tat. Tatsächlich kann man in der PowerShell viel erreichen, ohne sich mit dem zu Grunde liegenden .NET Framework zu beschäftigen. Dennoch: Wer alle Möglichkeiten der PowerShell nutzen will, braucht dann aber doch etwas Verständnis für objektorientiertes Programmieren und Erfahrung mit dem .NET Framework.

*Wenn Sie sich hier noch nicht auskennen, lesen Sie bitte zuerst in diesem Buch Anhang A Crashkurs „Objektorientierung“ und Anhang B Crashkurs „.NET Framework“.*

## 2.5 Anbindung an Klassenbibliotheken

Die Version 1.0 der PowerShell enthielt sehr viele Commandlets für die Pipelining-Infrastruktur, aber nur sehr wenige Befehle, die tatsächlich Bausteine des Betriebssystems in die Pipeline werfen. Prozesse, Systemdienste, Dateien, Zertifikate und Registrierungsdatenbankeinträge sind die magere Ausbeute beim ersten Blick in die Commandlet-Liste. Drei Commandlets eröffnen der PowerShell aber neue Dimensionen: `New-Object` (für .NET- und COM-Objekte) und `Get-WmiObject` bzw. `Get-CimInstance` (für WMI-Objekte). Seit Version 2.0 gibt es – zumindest in Verbindung mit neueren Betriebssystemen – mehr PowerShell-Befehle, die tatsächlich auf das Betriebssystem zugreifen.

**HINWEIS:** Die Option, nicht nur alle WMI-Klassen, sondern auch alle .NET-Klassen direkt benutzen zu können, ist Segen und Fluch zugleich. Ein Segen, weil dem Skriptentwickler damit mehr Möglichkeiten als jemals zuvor zur Verfügung stehen. Ein Fluch, weil nur der Skriptentwickler die PowerShell-Entwicklung richtig beherrschen kann, der auch das .NET Framework kennt. Um die Ausmaße von .NET zu beschreiben, sei die Menge der Klassen genannt. In .NET 2.0 waren es 6358, in .NET 3.5 sind es 10758, in .NET 4.7 sind es 13526.

## 2.6 PowerShell versus WSH

Administratoren fragen sich oft, wie sich die PowerShell im Vergleich zum Windows Script Host (WSH) positioniert, womit man neue Skripting-Projekte beginnen sollte und ob der WSH bald aus Windows verschwinden wird. Die folgende Tabelle trägt Fakten zusammen und bewertet auch die beiden Skripting-Plattformen.

**Tabelle 2.1** Vergleich WSH und Windows PowerShell bzw. PowerShell Core

| | Windows Script Host (WSH) | Windows PowerShell (WPS) | Windows PowerShell Core | PowerShell Core (PS Core) |
|---|---|---|---|---|
| Erstmals erschienen | 1998 | 2006 | 2017 | 2018 |
| Nummer der ersten Version | 1.0 | 1.0 | 5.1 | 6.0 |
| Aktueller Versionsstand | 5.8 | 5.1 | Core 5.1 | 6.0 |
| Läuft auf Windows-Betriebssystemen | Alle Windows-Betriebssysteme ab Windows 95/NT 4.0 | Version 1.0 ab Windows XP, Version 5.1 ab Windows 7 und Windows Server 2008 R2 | Windows Nano Server 2016 und Windows Nano Server 1709 | Windows ab Version 7, Windows Server ab Version 2008 R2 |

*(Fortsetzung nächste Seite)*

**Tabelle 2.1** Vergleich WSH und Windows PowerShell bzw. PowerShell Core *(Fortsetzung)*

| | Windows Script Host (WSH) | Windows PowerShell (WPS) | Windows PowerShell Core | PowerShell Core (PS Core) |
|---|---|---|---|---|
| Läuft auf anderen Betriebssysteme | Nein | Nein | Nein | diverse Linux-Distributionen, MacOS |
| Basis-Programmierframework | Component Object Model (COM) | .NET Framework | .NET Core | .NET Core |
| Derzeitiger Funktionsumfang | Sehr umfangreich | Funktionsumfang in Form von Commandlets abhängig vom Betriebssystem:<br>▪ nur wenige Commandlets vor Windows 7,<br>▪ bessere Unterstützung ab Windows 7,<br>▪ sehr umfangreich erst ab Windows 8 bzw. Windows Server 2012.<br>Wichtig: Auch ohne Commandlets steht auf den älteren Betriebssystemen aber ein hoher Funktionsumfang zur Verfügung, wenn man COM- oder .NET-Komponenten nutzt, was aber mehr Wissen voraussetzt. | Teilmenge von Windows PowerShell 5.1 | Teilmenge von Windows PowerShell 5.1 und wenige zusätzliche neue Funktionen |
| Weiterentwicklung der Laufzeitumgebung | Nein, nur noch beheben von Fehlern und Sicherheitslücken | Nein, nur noch beheben von Fehlern und Sicherheitslücken | Nein, nur noch beheben von Fehlern und Sicherheitslücken | Ja |
| Weiterentwicklung der Bibliotheken | Gering, gelegentlich erscheinen noch neue COM-Bibliotheken | Ja, Commandlet-Erweiterungen erscheinen immer wieder mit Microsoft-Produkten | Ja, Commandlet-Erweiterungen erscheinen immer wieder mit Microsoft-Produkten | Ja, Microsoft wird hier in den kommenden Jahren viel investieren |
| Weiterentwicklung der Werkzeuge | Nein | Ja | Ja | Ja |
| Basissyntax | Mächtig | Sehr mächtig | Sehr mächtig | Sehr mächtig |
| Direkte Skripting-Möglichkeiten | Alle COM-Komponenten mit IDispatch-Schnittstelle einschließlich WMI | Alle .NET-Komponenten, alle COM-Komponenten, alle WMI-Klassen | Alle .NET-Komponenten, alle COM-Komponenten, alle WMI-Klassen | Alle .NET Standard-Komponenten. COM und WMI nur unter Windows |

| | Windows Script Host (WSH) | Windows PowerShell (WPS) | Windows PowerShell Core | PowerShell Core (PS Core) |
|---|---|---|---|---|
| Skripting-Möglichkeiten über Wrapper | Alle Betriebssystemfunktionen | Alle Betriebssystemfunktionen | Viele Betriebssystemfunktionen | Viele Betriebssystemfunktionen |
| Werkzeuge von Microsoft | Scriptgeneratoren, Debugger, aber kein Editor | Integrated Scripting Environment (ISE), PowerShell Tools für Visual Studio, PowerShell-Erweiterung für VSCode | Integrated Scripting Environment (ISE), PowerShell Tools für Visual Studio, PowerShell-Erweiterung für VSCode | PowerShell-Erweiterung für VSCode, unter Windows auch ISE und PowerShell Tools für Visual Studio |
| Werkzeuge von Drittanbietern | Editoren, Debugger, Scriptgeneratoren | Editoren, Debugger, Scriptgeneratoren | Editoren, Debugger, Scriptgeneratoren | Bisher nur für Windows, siehe „Windows PowerShell“ |
| Einarbeitungsaufwand | Hoch | Mittel bis hoch (je nach Art der PowerShell-Nutzung) | Mittel bis hoch (je nach Art der PowerShell-Nutzung) | Mittel bis hoch (je nach Art der PowerShell-Nutzung) |
| Informationsverfügbarkeit | Hoch | Mittlerweile auch sehr hoch | Mittlerweile auch sehr hoch | Für die Nutzung unter Windows sehr hoch, für die anderen Betriebssysteme noch sehr gering |
| Startanwendung | cscript.exe und wscript.exe | powershell.exe | powershell.exe | pwsh.exe (Windows) bzw. pwsh (Linux und MacOS) |

**HINWEIS:** Hinweise zur Umstellung von WSH/VBScript auf die PowerShell finden Sie unter [TNET03].

# 3 Einzelbefehle der PowerShell

Die PowerShell kennt folgende Arten von Einzelbefehlen:

- Commandlets (inkl. Funktionen)
- Aliase
- Ausdrücke
- Externe Befehle
- Dateinamen

## 3.1 Commandlets

Ein „normaler“ PowerShell-Befehl heißt *Commandlet* (kurz: *Cmdlet*) oder *Funktion (Function)*. Eine Funktion ist eine Möglichkeit, in der PowerShell selbst wieder einen Befehl zu erstellen, der funktioniert wie ein Commandlet. Da die Unterscheidung zwischen Commandlets und Funktionen aus Nutzersicht zum Teil akademischer Art ist, erfolgt hier zunächst keine Differenzierung: Das Kapitel spricht allgemein von Commandlets und meint damit auch Funktionen.

### 3.1.1 Aufbau eines Commandlets

Ein Commandlet besteht typischerweise aus drei Teilen:

- einem Verb,
- einem Substantiv und
- einer (optionalen) Parameterliste.

Verb und Substantiv werden durch einen Bindestrich „-“ voneinander getrennt, die optionalen Parameter durch Leerzeichen. Daraus ergibt sich der folgende Aufbau:

```
Verb-Substantiv [-Parameterliste]
```

Die Groß- und Kleinschreibung ist bei den Commandlet-Namen nicht relevant.

### 3.1.2 Aufruf von Commandlets

Ein einfaches Beispiel ohne Parameter lautet:

```
Get-Process
```

Dieser Befehl liefert eine Liste aller laufenden Prozesse im System.

Ein zweites Beispiel ist:

```
Get-ChildItem
```

Dieser Befehl liefert Unterelemente des aktuellen Standorts. Meist ist der aktuelle Standort ein Dateisystempfad. In der PowerShell kann der aktuelle Standort aber auch in der Registrierungsdatenbank, dem Active Directory und vielen anderen (persistenten) Speichern liegen.

Ein drittes Beispiel ist:

```
Get-Service
```

Dieser Befehl liefert alle Windows-Systemdienste.

Das waren alles Commandlets, die Informationen liefern. Commandlets, die Aktionen ausführen (z. B. Prozesse beenden, Dateien löschen, Dienste anhalten), kommen in der Regel nicht ohne Parameter aus, da sie sonst ja global alle Dateien löschen würden. Das ist absichtlich nicht implementiert. Solche Befehle kommen daher erst im nächsten Unterkapitel vor.

**TIPP:** Die Tabulatorvervollständigung in der PowerShell-Konsole funktioniert bei Commandlets, wenn man das Verb und den Strich bereits eingegeben hat, z. B. `Export-`**Tab**. Auch Platzhalter kann man dabei verwenden. Die Eingabe `Get-?e*` **Tab** liefert `Get-Help` **Tab** `Get-Member` **Tab** `Get-Service`. Andere Editoren wie das ISE bieten auch IntelliSense-Eingabeunterstützung für Commandlet-Namen an.

**TIPP:** Commandlets, die mit dem Wort `Get-` beginnen, kann man abkürzen, indem man das `Get-` weglässt; also z. B. einfach `Service` statt `Get-Service` schreibt. Ob man dies so erlauben möchte, sollte das Unternehmen als Richtlinie festlegen.

### 3.1.3 Commandlet-Parameter

Durch Angabe eines Parameters können die Commandlets Informationen für die Befehlsausführung erhalten, z. B. ist bei `Get-Process` ein Filtern über den Prozessnamen möglich.

Durch

```
Get-Process  i*
```

werden nur diejenigen Prozesse angezeigt, deren Name auf das angegebene Muster (Name beginnt mit dem Buchstaben „i“) zutrifft:

Ein weiteres Beispiel für einen Befehl mit Parameter ist:

```
Get-ChildItem c:\daten
```

`Get-ChildItem` listet alle Unterobjekte des angegebenen Dateisystempfads *(c:\daten)* auf, also alle Dateien und Ordner unterhalb dieses Dateiordners.

Ein drittes Beispiel ist:

```
Stop-Service BITS
```

Dieser Befehl führt eine Aktion aus: Der Windows-Hintergrundübertragungsdienst (Background Intelligent Transfer Service – BITS) wird angehalten.

Ein viertes Beispiel ist:

```
Remove-Item c:\temp\*.log
```

Dieser Befehl löscht alle Dateien mit der Dateinamenserweiterung „log“ aus dem Ordner c:\temp.

Parameter werden als Zeichenkette aufgefasst – auch wenn sie nicht explizit in Anführungszeichen stehen. Die Anführungszeichen sind optional. Man muss Anführungszeichen um den Parameterwert nur dann verwenden, wenn Leerzeichen vorkommen, denn das Leerzeichen dient als Trennzeichen zwischen Parametern:

```
Get-ChildItem "C:\Program Files"
```

Einige Commandlets erlauben für einen Parameter nicht nur einen einzelnen Wert, sondern auch eine Menge von Werten. Die Einzelwerte sind dann durch ein Komma zu trennen.

**Beispiel:** Prozesse, die mit dem Buchstaben a beginnen oder enden oder mit x beginnen oder enden

```
Get-Process  "a*","*a","x*","*x"
```

```
PS T:\> Get-Process "a*","*a","x*","*x"

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    629      29    21672      26484       1,27    1200   4 ApplicationFrameHost
    140       9     1420       1928       0,05    4276   0 armsvc
    137       8     1484       1976       0,11    3192   0 atiesrxx
   1259      68    75644      11056  32.389,55    4356   0 AVKProxy
    896      38   155380     155808   4.404,88    3096   0 AVKWCtlx64
    993      90   210684     244996      40,86   13128   4 firefox
```

**Bild 3.1** Get-Process mit einer Liste von Namen

Commandlets haben aber in der Regel nicht nur einen, sondern zahlreiche Parameter, die durch Position oder einen Parameternamen voneinander unterschieden werden. Ohne die Verwendung von Parameternamen werden vordefinierte Standardattribute belegt, d. h., die Reihenfolge ist entscheidend.

**Beispiel:** Auflisten von Dateien in einem Dateisystempfad, die eine bestimmte Datennamenserweiterung besitzen. Dies erfüllt der Befehl:

```
Get-ChildItem C:\temp *.doc
```

Wenn ein Commandlet mehrere Parameter besitzt, ist die Reihenfolge der Parameter entscheidend oder der Nutzer muss die Namen der Parameter mit angeben. Bei der Angabe von Parameternamen kann man die Reihenfolge der Parameter ändern:

```
Get-ChildItem -Filter *.doc -Path C:\temp
```

Alle folgenden Befehle sind daher gleichbedeutend:

```
Get-ChildItem C:\temp *.doc
Get-ChildItem -Path C:\temp -Filter *.doc
Get-ChildItem -Filter *.doc -Path C:\temp
```

Hingegen ist Folgendes falsch und funktioniert nicht wie gewünscht, weil die Parameter nicht benannt sind und die Reihenfolge falsch ist:

```
Get-ChildItem *.doc C:\temp
```

Diesen Versuch beantwortet die PowerShell mit einer Fehlermeldung („Das zweite Pfadfragment darf kein Laufwerk oder UNC-Name sein.“) in roter Schrift (siehe Bild 3.1).

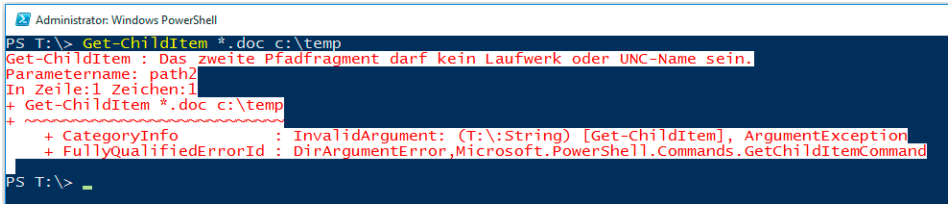



**Bild 3.2** Fehlermeldung bei falscher Parameterreihenfolge

Schalter-Parameter (engl. Switch) sind Parameter, die keinen Wert haben. Durch die Verwendung des Parameternamens wird die Funktion aktiviert, z. B. das rekursive Durchlaufen durch einen Dateisystembaum mit `-recurse`:

```
Get-ChildItem x:\demo\powershell -recurse
```

**TIPP:** Wenn man einen Schalter deaktivieren möchte, weil er im Standard aktiv ist oder weil man sehr explizit darauf hinweisen möchte, dass er nicht aktiv sein soll, kann man `$false` mit Doppelpunkt getrennt angeben, z. B.

```
Get-ChildItem x:\demo\powershell -recurse:$false
```

Parameter können berechnet, d. h. aus Teilzeichenketten zusammengesetzt sein, die mit einem Pluszeichen verbunden werden. (Dies macht insbesondere Sinn in Zusammenhang mit Variablen, die aber erst später in diesem Buch eingeführt werden.)

Der folgende Ausdruck führt jedoch nicht zum gewünschten Ergebnis, da auch hier das Trennzeichen vor und nach dem + ein Parametertrenner ist.

```
Get-ChildItem "c:\" + "Windows" *.dll -Recurse
```

Auch ohne die beiden Leerzeichen vor und nach dem + geht es nicht. In diesem Fall muss man durch eine runde Klammer dafür sorgen, dass die Berechnung erst ausgeführt wird:

```
Get-ChildItem ("c:\" + "Windows") *.dll -Recurse
```

Es folgt dazu noch ein Beispiel, bei dem Zahlen berechnet werden. Der folgende Befehl liefert den Prozess mit der ID 2900:

```
Get-Process -id (2800+100)
Get-Service -exclude "[k-z]*"
```

zeigt nur diejenigen Systemdienste an, deren Name nicht mit den Buchstaben „k“ bis „z“ beginnt.

Auch mehrere Parameter können der Einschränkung dienen. Der folgende Befehl liefert nur die Benutzereinträge aus einem bestimmten Active-Directory-Pfad. (Das Beispiel setzt die Installation der PSCX voraus.)

```
Get-ADObject -dis "LDAP://D142/ou=agents,DC=FBI,DC=net" -class user
```

**TIPP:** Tabulatorvervollständigung klappt auch bei Parametern. Versuchen Sie einmal folgende Eingabe an der PowerShell-Konsole: `Get-ChildItem -Tab`

### 3.1.4 Platzhalter bei den Parameterwerten

An vielen Stellen sind Platzhalter bei den Parameterwerten erlaubt.

Ein Stern steht für beliebig viele Zeichen. Eine Liste aller Prozesse, die mit einem „i“ anfangen, erhält man so:

```
Get-Process i*
```

Eine Liste aller Prozesse, die mit einem „i“ anfangen und auf „ore“ enden, erhält man so:

```
Get-Process i*ore
```

Ein Fragezeichen steht für genau ein beliebiges Zeichen. Eine Liste aller Prozesse, die mit einem „v“ anfangen, gefolgt von einem einzigen beliebigen Zeichen und auf „mms“ enden, erhält man so:

```
Get-Process v?mms
```

Eine eckige Klammer steht für genau ein Zeichen aus einer Auswahl. Alle Prozesse, die mit s oder t anfangen, erhält man so:

```
Get-Process [st]*
```

Alle Prozesse, die mit s oder t anfangen und bei denen dann ein v oder f folgt, erhält man so:

```
Get-Process [st][vf]*
```

```
PS C:\> get-process [st][vf]*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1439     162    29196      44776   581              484 svchost
    284      14     5448       5772   308              568 svchost
   2958     107   315244      66196   608              576 svchost
    417      16     5728      10464    53              892 svchost
    748      19    14500      17492    65              948 svchost
    747      45    23100      27612   152             1028 svchost
    819      35    42244      40380   123             1084 svchost
   1403      77    83400      82220  1788             1324 svchost
    462      68    36844      32792   399             1508 svchost
    155      13     5144       5536    74             2224 svchost
    261      14     6056       5832    50             2356 svchost
    152      12     2560       8480    48             3340 svchost
    177      16     7560       8372    54             3648 svchost
    101       8     1656       2068    23             5196 svchost
    468      28     7308      11324    87             5628 svchost
    614      45    42160      27224   397          2,73  7428 TfsCommandRunnerSvr
    608      32    26312      25776   232         82,68  1496 TfsComProviderSvr
    530      30    22540      21616   218          1,14 10700 TfsComProviderSvr
```

**Bild 3.3** Beispiele für das Ergebnis des obigen Befehls mit Platzhaltern

### 3.1.5 Abkürzungen für Parameter

Parameternamen dürfen abgekürzt werden, solange sie noch eindeutig sind.
Statt

```
Get-ChildItem -Filter *.txt -Path C:\temp
```

darf man schreiben

```
Get-ChildItem -Fi *.txt -Pa C:\temp
```

Nicht möglich ist in diesem Fall die Reduzierung auf einen Buchstaben:

```
Get-ChildItem -F *.txt -P C:\temp
```

Darauf reagiert die PowerShell mit der Fehlermeldung

```
Der Parameter kann nicht verarbeitet werden, da der Parametername "F" nicht eindeutig
ist. Mögliche Übereinstimmungen:  -Filter -Force
```

und

```
Der Parameter kann nicht verarbeitet werden, da der Parametername "P" nicht eindeutig
ist. Mögliche Übereinstimmungen:  -Path -PipelineVariable
```

**ACHTUNG:** Bitte beachten Sie aber, dass abgekürzte Parameter auch eine Gefahr bedeuten: Was heute eine eindeutige Abkürzung ist, könnte in einer zukünftigen Version doppeldeutig sein, wenn Microsoft weitere Parameter zu einem Commandlet ergänzt. Tatsächlich gab es in der Vergangenheit auch schon kuriose Fälle, dass die Abkürzungen in verschiedenen Windows-Installationen verschieden interpretiert wurden, wie die nachstehenden Bildschirmabbildungen beweisen. Zudem sind abgekürzte Parameter nicht so „sprechend" wie die Langparameter. Für abgekürzte Parameter spricht aber, dass Befehle dadurch kürzer und übersichtlicher werden.

Trotz allem werden Sie auch abgekürzte Parameter in diesem Buch finden, da der Autor dieses Buchs eben auch ein Mensch ist, der sich im Alltag manche Tipparbeit gerne erspart.


**Bild 3.4** Verhalten auf Windows Server 2008 R2 und Windows 8 mit PowerShell 3.0

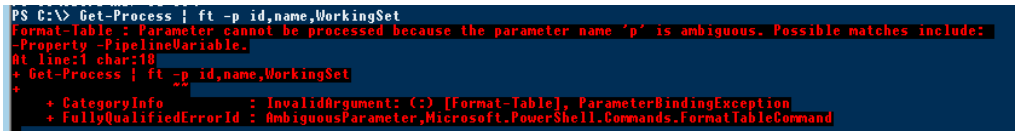



**Bild 3.5** Verhalten auf Windows 7 und Windows 8.1 mit PowerShell 4.0

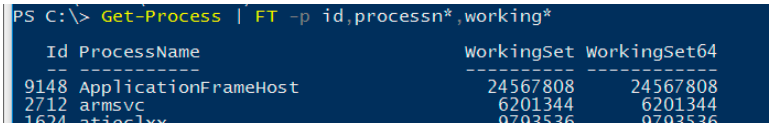



**Bild 3.6** Verhalten auf allen Windows 10 und Windows Server 2012 R2 mit PowerShell 5.x

### 3.1.6 Allgemeine Parameter (Common Parameters)

Es gibt einige Parameter, die in vielen (aber nicht allen) Commandlets vorkommen. Es folgt eine vollständige Liste dieser Parameter. Eine genauere Beschreibung folgt aber aus didaktischen Gründen an geeigneter Stelle im Buch, da viele allgemeine Parameter mit dem Pipelining und der Fehlerbehandlung zu tun haben, die erst in späteren Kapiteln besprochen wird.

- `-Force`: Eine Aktion wird erzwungen, z. B. eine Datei wird mit `Remove-Item` gelöscht, obwohl die Datei einen Schreibschutz gesetzt hat. Ein weiteres Beispiel: `Remove-SmbShare` fragt immer vor dem Löschen nach, wenn `–force` nicht gesetzt ist.

- `-Whatif` („Was wäre wenn“): Die Aktion wird nicht ausgeführt, es wird nur ausgegeben, was passieren würde, wenn man die Aktion ausführt. Das ist z. B. in einem Befehl mit Platzhaltern wie dem Folgenden sinnvoll, damit man weiß, welche Dienste nun gestoppt würden:

```
Get-Service | Where {$_.servicename -like "A*"}
| Foreach { stop-service $_.servicename -whatif}
```

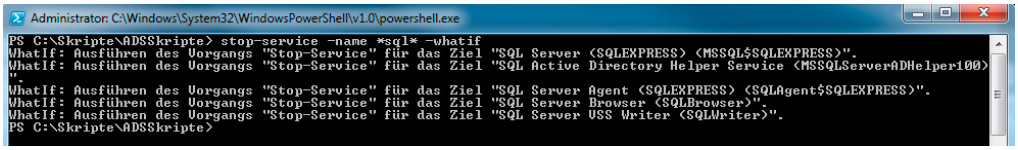



**Bild 3.7** Operationen mit Platzhaltern können schlimme Konsequenzen haben – whatif zeigt, welche Dienste betroffen wären.

- `-Confirm`: Der Benutzer erhält eine Nachfrage für jede Änderungsaktion (siehe Bildschirmabbildung), z. B.

```
get-service | where {$_.servicename -like "A*"}
| foreach { stop-service $_.servicename –confirm }.
```

Innerhalb der Nachfrage kann der Benutzer in einen Suspend-Modus gehen, in dem er andere Befehle eingeben kann, z. B. um zu prüfen, ob er nun ja oder nein antworten will. Der Suspend-Modus wird mit drei Pfeilen >>> angezeigt und ist durch exit zu verlassen (siehe Bildschirmabbildung).

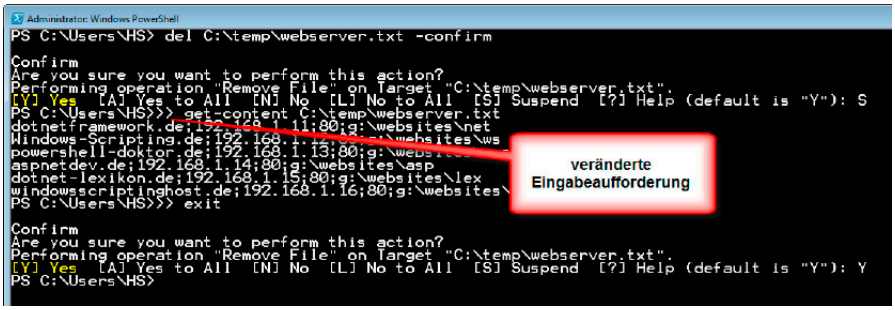



**Bild 3.8** Confirm und Suspend

- `-ErrorAction` (abgekürzt `-ea`) und `-WarningAction` (`-wa`): Festlegung, wie ein Skript sich verhalten soll, wenn es auf einen Fehler trifft. Dieser Parameter wird im Abschnitt 7.19 „Fehlerbehandlung“ näher erklärt.
- `-Verbose`: Das Commandlet liefert eine detaillierte Bildschirmausgabe.
- `-Debug`: Das Commandlet liefert eine sehr detaillierte Bildschirmausgabe.
- `-OutVariable`: Das Commandlet liefert alle Objekte nicht nur in die Pipeline, sondern legt sie zusätzlich auch in einer Variablen ab.
- `-PipelineVariable`: Das Commandlet liefert das aktuelle Objekt nicht nur in die Pipeline, sondern legt es zusätzlich auch in einer Variablen ab.

- `-ErrorAction`: Festlegung, wie sich das Commandlet bei Fehlern verhält
- `-ErrorVariable`: speichert eine Fehlermeldung des Commandlets zusätzlich in einer Variablen
- `-WarningAction`: Festlegung, wie sich das Commandlet bei Warnungen verhält. Der Standard ist „continue“, was bedeutet, dass die Meldung ausgegeben wird. Mit „silentlycontinue“ kann die Ausgabe unterdrückt werden. Mit „stop“ wird ein Befehl nach der Warnung abgebrochen. Mit „inquire“ fragt die PowerShell nach, wie fortzufahren ist.
- `-WarningVariable`: speichert eine Warnung des Commandlets zusätzlich in einer Variablen
- `-OutBuffer`: stellt ein, dass die angegebene Anzahl von Objekten in der Pipeline gepuffert werden sollen, bevor sie in der Pipeline weitergegeben werden. Normalerweise werden alle Objekte sofort in der Pipeline weitergegeben.

**ACHTUNG:** Leider beachten nicht alle Commandlets alle allgemeinen Parameter. Erschwerend kommt hinzu, dass sie keine Fehlermeldung liefern, sondern den Parameter einfach ignorieren. Ein Beispiel ist `New-SmbShare` zum Anlegen einer Dateisystemfreigabe. Die folgenden Befehle werden trotz `-whatif` bzw. `-confirm` sofort und ohne Nachfrage ausgeführt.

```
New-SmbShare -Name Temp -Path c:\temp -WhatIf
New-SmbShare -Name Temp -Path c:\temp -confirm
```

Sie werden sich fragen, warum dies so ist. Das Fehlverhalten liegt hier bei dem Entwickler des Commandlets. Jeder Commandlet-Entwickler muss daran denken, die allgemeinen Parameter zu behandeln. Denkt er nicht daran, sind die Nutzer seines Commandlets die Leidtragenden. Es wäre natürlich besser, wenn Microsoft mit seiner Programmierschnittstelle für Commandlets die Commandlet-Entwickler zwingen würde, die Parameter zu behandeln oder zumindest eine Fehlermeldung zu liefern, wenn man die Parameter einsetzt. Leider hat Microsoft diesen Vorschlag bisher nicht aufgegriffen – auch wenn Microsoft ja sehr offensichtlich nicht mal seine eigenen Commandlet-Entwickler im Griff hat.

**ACHTUNG:** Leider gibt es bei den PowerShell-Commandlets, die gravierende Aktionen ausführen, einige Unterschiede im Grundverhalten und in der Verwendung der obigen Commandlets. Einige Commandlets führen im Standard die Aktion aus (z. B. `Remove-Item`). Andere Commandlets (z. B. `Remove-ADUser` und `Remove-SmbShare`) fragen immer nach vor dem Löschen. Das ist bei automatisierten Skripten natürlich unsinnig und daher gibt es auch eine Möglichkeit, diesen Commandlets das abzugewöhnen. Diese sieht jedoch oftmals verschieden aus. Bei `Remove-ADUser` muss man `–confirm :$false` als Parameter angeben; bei `Remove-SmbShare` ist es hingegen ein `-force`. Schade, dass Microsoft hier nicht einheitlich sein konnte.

### Standardvorgaben für allgemeine Parameter

In den eingebauten Variablen `$WhatIfPreference`, `$VerbosePreference`, `$DebugPreference`, `$ConfirmPreference` und `$ErrorActionPreference` ist festgelegt, wie sich die PowerShell im Standard in Bezug auf -WhatIf, -Verbose, `-Debug`, `-Confirm` und `-ErrorAction` verhält. Dort ist hinterlegt:

- `WhatIfPreference: False`
- `VerbosePreference: SilentlyContinue`
- `DebugPreference: SilentlyContinue`
- `ErrorActionPreference: Continue`
- `ConfirmPreference: High`

Variablen werden erst später in diesem Buch (Kapitel 7 „PowerShell-Skriptsprache“) behandelt. An dieser Stelle soll aber schon mit einem Beispiel gezeigt werden, wie man `$WhatIfPreference` auf `$true` setzt und damit erreicht, dass alle Commandlets, die `-whatif` unterstützen, nun nur noch sagen, was sie machen würden – zumindest solange man nicht explizit `-whatif:$false` als Parameter angibt.

Ausgabe der aktuellen Einstellung von `$WhatIfPreference`. Sollte `$false` sein

```
Write-host "WhatIfPreference = $WhatIfPreference" -ForegroundColor Yellow
```

Neustart des Dienstes wird tatsächlich ausgeführt

```
Restart-Service BITS -WhatIf -Verbose
```

Nun `$WhatIfPreference` aktivieren

```
$WhatIfPreference = $true
```

Ausgabe der aktuellen Einstellung von `$WhatIfPreference`. Sollte `$true` sein

```
Write-host "WhatIfPreference = $WhatIfPreference" -ForegroundColor Yellow
```

Neustart des Dienstes wird NICHT ausgeführt

```
Restart-Service BITS -Verbose
```

Neustart des Dienstes wird tatsächlich ausgeführt

```
Restart-Service BITS -WhatIf:$false -Verbose
```

Nun `$WhatIfPreference` zurücksetzen

```
$WhatIfPreference = $false
```

### 3.1.7 Dynamische Parameter

Einige Commandlets besitzen die Fähigkeit, verschiedene Parameter abhängig von bereits eingegebenen Parametern anzubieten.

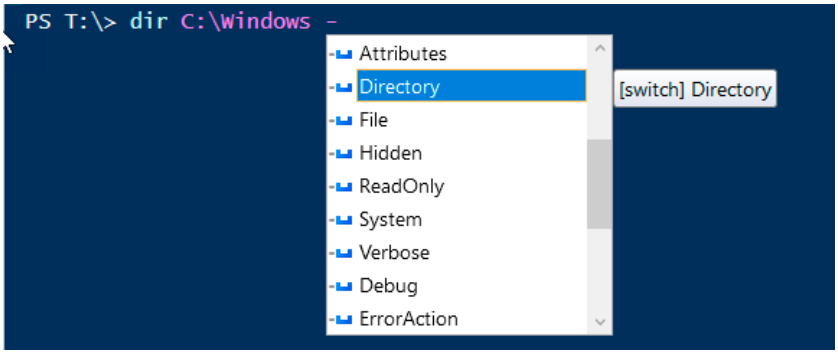



**Bild 3.9** Get-ChildItem (alias dir) in Verbindung mit einem Dateisystempfad

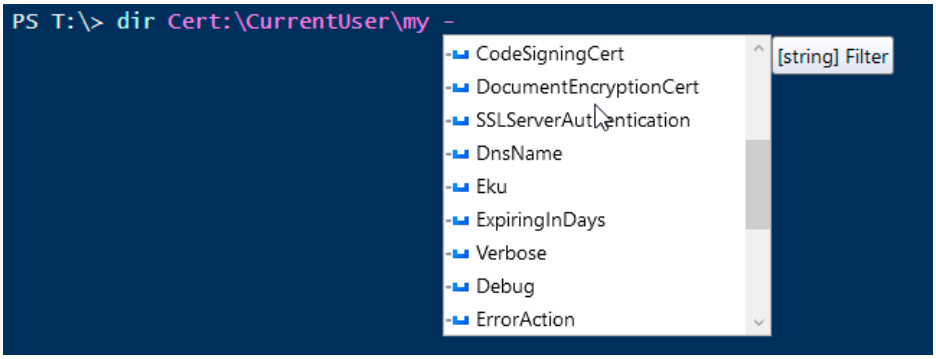



**Bild 3.10** Get-ChildItem (alias dir) in Verbindung mit einem Pfad im Zertifikatsspeicher

### 3.1.8 Zeilenumbrüche

Wenn man die **Eingabe**-Taste drückt, wird ein PowerShell direkt ausgeführt. Möchte man einen Befehl über mehrere Zeilen erstrecken, muss man die unvollständige Zeile mit dem Gravis (Accent Grave) [`] beenden.

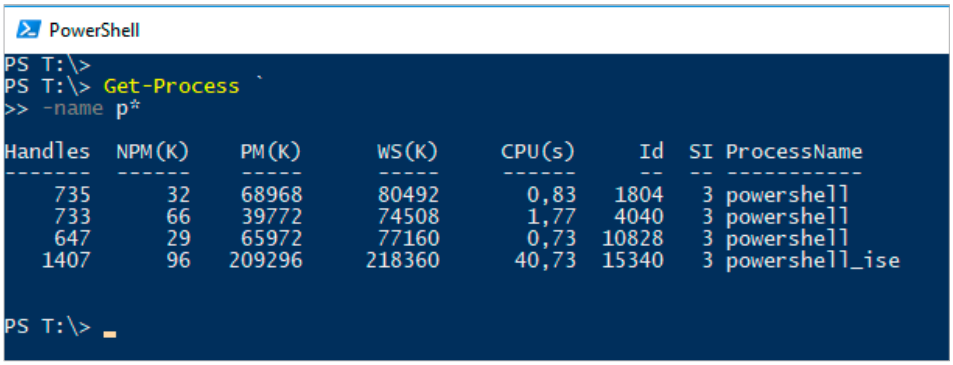



**Bild 3.11** Einsatz des Gravis für Zeilenumbrüche im Befehl

**TIPP:** In der PowerShell-Konsole kann man **SHIFT** + **EINGABE** drücken. Auch dann zeigt die Standardkonsole mit >>> an, dass weitere Eingaben erwartet werden. Allerdings wird dann ein eigenständiger Befehl erwartet und nicht der vorherige fortgesetzt!

### 3.1.9 PowerShell-Module

Schon seit PowerShell 2.0 sind die Commandlets und Funktionen in Modulen organisiert. Während der Benutzer in PowerShell 2.0 ein Modul noch explizit mit `Import-Module` aktivieren musste, bevor man die Befehle aus dem Modul nutzen konnte, erledigt dies die PowerShell seit Version 3.0 bei Bedarf automatisch (Module Auto-Loading). Sowohl Konsole als auch ISE zeigen alle verfügbaren Commandlets und Funktionen aller vorhandenen Module in der Vorschlagsliste und beim Aufruf von `Get-Command` bereits an. Der eigentliche Import des Moduls erfolgt dann beim ersten Aufruf eines Befehls aus einem Modul.

In der PowerShell sind auch alle Kernbefehle der PowerShell in Modulen organisiert, diese zeigt die folgende Tabelle.

**Tabelle 3.1** Die vier wichtigsten Module der PowerShell mit Beispielen für Commandlets in diesem Modul

| Modul | Beispiele für Commandlets in diesem Modul |
|---|---|
| Microsoft.PowerShell.Diagnostics | Get-WinEvent, Get-Counter, Import-Counter, Export-Counter ... |
| Microsoft.PowerShell.Management | Add-Content, Clear-Content, Clear-ItemProperty, Join-Path, Get-Process, Get-Service ... |
| Microsoft.PowerShell.Security | Get-Acl, Set-Acl, Get-PfxCertificate, Get-Credential ... |
| Microsoft.PowerShell.Utility | Format-List, Format-Custom, Format-Table, Format-Wide, Where-Object ... |

### 3.1.10 Prozessmodell

Die PowerShell erzeugt beim Start einen einzigen Prozess. In diesem Prozess laufen alle ausgeführten Commandlets. Dies ist ein Unterschied zum DOS-ähnlichen Windows-Kommandozeilenfenster, bei dem die ausführbaren Dateien (*.exe*) in eigenen Prozessen laufen. Es ist in der PowerShell aber auch möglich, Hintergrundaufgaben auszuführen (siehe Kapitel 25 „Hintergrundaufträge“).

**TIPP:** Mit STRG+C kann man einen laufenden Befehl in der PowerShell abbrechen.

### 3.1.11 Aufruf von Commandlets aus anderen Prozessen heraus

PowerShell-Commandlets kann man aus einem beliebigen Prozess heraus aufrufen, indem man powershell.exe aufruft und das Commandlet als Parameter übergibt.

**Beispiel:** `powershell.exe "get-service a*"`

Damit die Parameter des Commandlets dem Commandlet und nicht powershell.exe zugeordnet werden, muss man das Commandlet und seine Parameter in Anführungszeichen setzen. Falls der PowerShell-Commandlet-Parameter seinerseits Anführungszeichen erfordert, muss man dafür einfache Anführungszeichen verwenden:

```
powershell.exe "get-service 'a*'"
```

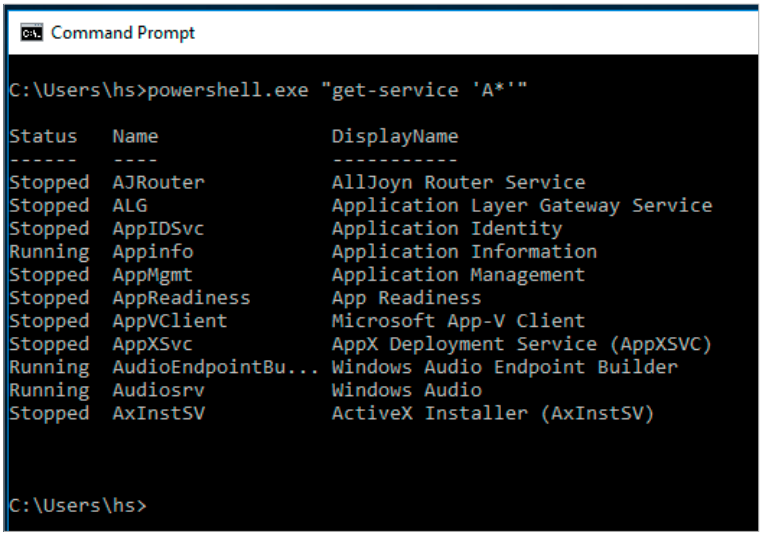



**Bild 3.12** Aufruf eines PowerShell-Commandlets aus einer klassischen Windows-Konsole (CMD) heraus mit powershell.exe (im Bild in Windows 10)

### 3.1.12 Namenskonventionen

Man beachte, dass bei den Commandlets das Substantiv im Singular steht, auch wenn eine Menge von Objekten abgerufen wird. Das Ergebnis muss nicht immer eine Objektmenge sein. Beispielsweise liefert

```
Get-Location
```

nur ein Objekt mit dem aktuellen Pfad.

Mit

```
Set-Location c:\windows
```

wechselt man den aktuellen Pfad. Diese Operation liefert gar kein Ergebnis.

**HINWEIS:** Die Groß- und Kleinschreibung der Commandlet-Namen und der Parameternamen ist irrelevant.

Gemäß der PowerShell-Konventionen soll es nur eine begrenzte Menge wiederkehrender Verben geben: `Get`, `Set`, `Add`, `New`, `Remove`, `Clear`, `Push`, `Pop`, `Write`, `Export`, `Select`, `Sort`, `Update`, `Start`, `Stop`, `Invoke` usw. Außer diesen Basisoperationen gibt es auch Ausgabekommandos mit Verben wie `Out` und `Format`. Auch Bedingungen werden durch diese Syntax abgebildet (`Where-Object`).

## 3.2 Aliase

Durch sogenannte Aliase kann die Eingabe von Commandlets verkürzt werden. So ist `ps` als Alias für `Get-Process` oder `help` für `Get-Help` vordefiniert. Statt `Get-Process i*` kann also auch geschrieben werden: `ps i*`.

**HINWEIS:** Manche PowerShell-Experten betrachten den Einsatz von Aliasen als schlechten Stil, der die Lesbarkeit von PowerShell-Skripten erschwert. Auf der anderen Seite ersparen Aliase eben Tipparbeit. Ob man vordefinierte und ggf. auch selbst definierte PowerShell-Aliase erlauben möchte, sollte man im Unternehmen als Richtlinie festlegen.

### 3.2.1 Aliase auflisten

Durch `Get-Alias` (oder den entsprechenden Alias `aliases`) erhält man eine Liste aller vordefinierten Abkürzungen in Form von Instanzen der Klasse `System.Management.Automation.AliasInfo`.

Durch Angabe eines Namens bei `Get-Alias` erhält man die Bedeutung eines Alias:

```
Get-Alias pgs
```

Möchte man zu einem Commandlet alle Aliase wissen, muss man allerdings schreiben:

```
Get-Alias | Where-Object { $_.definition -eq "Get-Process" }
```

Dies erfordert schon den Einsatz einer Pipeline, die erst im nächsten Kapitel besprochen wird.

**Tabelle 3.2** Vordefinierte Aliase in der PowerShell 5.1

| Alias | Commandlet |
|---|---|
| % | ForEach-Object |
| ? | Where-Object |
| ac | Add-Content |
| asnp | Add-PSSnapIn |
| cat | Get-Content |
| cd | Set-Location |
| chdir | Set-Location |
| clc | Clear-Content |
| clear | Clear-Host |
| clhy | Clear-History |
| cli | Clear-Item |
| clp | Clear-ItemProperty |
| cls | Clear-Host |
| clv | Clear-Variable |
| cnsn | Connect-PSSession |
| compare | Compare-Object |
| copy | Copy-Item |
| cp | Copy-Item |
| cpi | Copy-Item |
| cpp | Copy-ItemProperty |
| cvpa | Convert-Path |
| dbp | Disable-PSBreakpoint |
| del | Remove-Item |

*(Fortsetzung nächste Seite)*

**Tabelle 3.2** Vordefinierte Aliase in der PowerShell 5.1 *(Fortsetzung)*

| Alias | Commandlet |
|---|---|
| diff | Compare-Object |
| dir | Get-ChildItem |
| dnsn | Disconnect-PSSession |
| ebp | Enable-PSBreakpoint |
| echo | Write-Output |
| epal | Export-Alias |
| epcsv | Export-CSV |
| epsn | Export-PSSession |
| erase | Remove-Item |
| etsn | Enter-PSSession |
| exsn | Exit-PSSession |
| fc | Format-Custom |
| fl | Format-List |
| foreach | ForEach-Object |
| ft | Format-Table |
| fw | Format-Wide |
| gal | Get-Alias |
| gbp | Get-PSBreakpoint |
| gc | Get-Content |
| gci | Get-ChildItem |
| gcm | Get-Command |
| gcs | Get-PSCallStack |
| gdr | Get-PSDrive |
| ghy | Get-History |
| gi | Get-Item |
| gjb | Get-Job |
| gl | Get-Location |
| gm | Get-Member |
| gmo | Get-Module |
| gp | Get-ItemProperty |
| gps | Get-Process |
| group | Group-Object |
| gsn | Get-PSSession |
| gsnp | Get-PSSnapIn |
| gsv | Get-Service |
| gu | Get-Unique |
| gv | Get-Variable |

| Alias | Commandlet |
|---|---|
| gwmi | Get-WmiObject |
| h | Get-History |
| history | Get-History |
| icm | Invoke-Command |
| iex | Invoke-Expression |
| ihy | Invoke-History |
| ii | Invoke-Item |
| ipal | Import-Alias |
| ipcsv | Import-CSV |
| ipmo | Import-Module |
| ipsn | Import-PSSession |
| irm | Invoke-RestMethod |
| ise | powershell_ise.exe |
| iwmi | Invoke-WMIMethod |
| iwr | Invoke-WebRequest |
| kill | Stop-Process |
| lp | Out-Printer |
| ls | Get-ChildItem |
| man | help |
| md | mkdir |
| measure | Measure-Object |
| mi | Move-Item |
| mount | New-PSDrive |
| move | Move-Item |
| mp | Move-ItemProperty |
| mv | Move-Item |
| nal | New-Alias |
| ndr | New-PSDrive |
| ni | New-Item |
| nmo | New-Module |
| npssc | New-PSSessionConfigurationFile |
| nsn | New-PSSession |
| nv | New-Variable |
| ogv | Out-GridView |
| oh | Out-Host |
| popd | Pop-Location |
| ps | Get-Process |

*(Fortsetzung nächste Seite)*

**Tabelle 3.2** Vordefinierte Aliase in der PowerShell 5.1 *(Fortsetzung)*

| Alias | Commandlet |
|---|---|
| pushd | Push-Location |
| pwd | Get-Location |
| r | Invoke-History |
| rbp | Remove-PSBreakpoint |
| rcjb | Receive-Job |
| rcsn | Receive-PSSession |
| rd | Remove-Item |
| rdr | Remove-PSDrive |
| ren | Rename-Item |
| ri | Remove-Item |
| rjb | Remove-Job |
| rm | Remove-Item |
| rmdir | Remove-Item |
| rmo | Remove-Module |
| rni | Rename-Item |
| rnp | Rename-ItemProperty |
| rp | Remove-ItemProperty |
| rsn | Remove-PSSession |
| rsnp | Remove-PSSnapin |
| rujb | Resume-Job |
| rv | Remove-Variable |
| rvpa | Resolve-Path |
| rwmi | Remove-WMIObject |
| sajb | Start-Job |
| sal | Set-Alias |
| saps | Start-Process |
| sasv | Start-Service |
| sbp | Set-PSBreakpoint |
| sc | Set-Content |
| select | Select-Object |
| set | Set-Variable |
| shcm | Show-Command |
| si | Set-Item |
| sl | Set-Location |
| sleep | Start-Sleep |
| sls | Select-String |
| sort | Sort-Object |

| Alias | Commandlet |
|---|---|
| sp | Set-ItemProperty |
| spjb | Stop-Job |
| spps | Stop-Process |
| spsv | Stop-Service |
| start | Start-Process |
| sujb | Suspend-Job |
| sv | Set-Variable |
| swmi | Set-WMIInstance |
| tee | Tee-Object |
| trcm | Trace-Command |
| type | Get-Content |
| where | Where-Object |
| wjb | Wait-Job |
| write | Write-Output |

### 3.2.2 Neue Aliase anlegen

Einen neuen Alias definiert der Nutzer mit `Set-Alias` oder `New-Alias`, z. B.:

```
Set-Alias procs Get-Process
New-Alias procs Get-Process
```

Der Unterschied zwischen `Set-Alias` und `New-Alias` ist marginal: `New-Alias` erstellt einen neuen Alias und liefert einen Fehler, wenn der zu vergebende Alias schon existiert. `Set-Alias` erstellt einen neuen Alias oder überschreibt einen Alias, wenn der zu vergebende Alias schon existiert. Mit dem Parameter `-description` kann man jeweils auch einen Beschreibungstext setzen.

Man kann einen Alias nicht nur für Commandlets, sondern auch für klassische Anwendungen vergeben, z. B.:

```
Set-Alias np notepad.exe
```

**ACHTUNG:** Beim Anlegen eines Alias wird nicht geprüft, ob das zugehörige Commandlet bzw. die Anwendung überhaupt existiert. Der Fehler würde erst beim Aufruf des Alias auftreten.

Beim Anlegen eines Alias muss man zudem aufpassen, dass man keine bestehenden Namen überschreibt, denn Aliase haben Priorität. Wenn man `Set-Alias notepad dir` eingibt, führt ab dann die Eingabe von `notepad` nicht mehr zu `notepad.exe`, sondern zum Commandlet `Get-ChildItem` (für das `dir` ein Alias ist). `notepad` ist dann also ein Alias für einen Alias.

Man kann in Aliasdefinitionen keinen Parameter mit Werten vorbelegen. Möchten Sie zum Beispiel definieren, dass die Eingabe von „Temp“ die Aktion „Get-ChildItem c:\Temp“ ausführt, brauchen Sie dafür eine Funktion. Mit einem Alias geht das nicht.

```
Function Temp { Get-ChildItem c:\temp }
```

Funktionen werden später (siehe Kapitel 6 *„PowerShell-Skripte“*) noch ausführlich besprochen. Die PowerShell enthält zahlreiche vordefinierte Funktionen, z. B. `c:`, `d:`, `e:` sowie `mkdir` und `help`.

Die neu definierten Aliase gelten jeweils nur für die aktuelle Instanz der PowerShell-Konsole. Man kann die eigenen Alias-Definitionen exportieren mit `Export-Alias` und später wieder importieren mit `Import-Alias`. Als Speicherformate stehen das CSV-Format und das PowerShell-Skriptdateiformat (*.ps1* – siehe spätere Kapitel) zur Verfügung. Bei dem ps1-Format ist zum späteren Reimport der Datei das Skript mit dem Punktoperator (engl. „Dot Sourcing“) aufzurufen.

| | Dateiformat CSV | Dateiformat .ps1 |
|---|---|---|
| Speichern | `Export-Alias c:\meinealias.csv` | `Export-Alias c:\meinealias.ps1 -as script` |
| Laden | `Import-Alias c:\meinealias.csv` | `. c:\meinealias.ps1` |

Die Anzahl der Aliase ist im Standard auf 4096 beschränkt. Dies kann durch die Variable `$MaximumAliasCount` geändert werden.

**HINWEIS:** Aliase entfernen aus der aktuellen PowerShell kann man mit Remove-Item alias:\AliasName, also z.B. Remove-Item alias:\np. Ab PowerShell Core 6.0 gibt es auch das Commandlet Remove-Alias zu diesem Zweck: Remove-Alias np.

### 3.2.3 Aliase für Eigenschaften

Aliase sind auch auf Ebene von Eigenschaften definiert. So kann man statt

```
Get-Process processname, workingset
```

auch schreiben:

```
Get-Process  name, ws
```

Diese Aliase der Attribute sind definiert in der Datei *types.ps1xml* im Installationsordner der PowerShell.


**Bild 3.13** types.ps1xml

**ACHTUNG:** Die types.ps1xml-Datei wird ab PowerShell 5.1 nicht mehr von der PowerShell verwendet, da das Einlesen der Datei die Startgeschwindigkeit der PowerShell-Konsolen negativ beeinflusst hat. Die Informationen liegen nun im C#-Code der Commandlets vor. Die types.ps1xml ist noch für den PowerShell 2.0-Kompatibilitätsmodus vorhanden.

## 3.3 Ausdrücke

Ebenfalls als Befehl direkt in die PowerShell eingeben kann man Ausdrücke, z. B. mathematische Ausdrücke wie

```
10* (8 + 6)
```

oder Zeichenkettenausdrücke wie

```
"Hello "+ " " + "World"
```

Microsoft spricht hier vom Expression Mode der PowerShell im Kontrast zum Command Mode, der verwendet wird, wenn man

```
Write-Output 10* (8 + 6)
```

aufruft.

Die PowerShell kennt zwei Verarbeitungsmodi für Befehle: einen Befehlsmodus (Command Mode) und einen Ausdrucksmodus (Expression Mode). Im Befehlsmodus werden alle Eingaben als Zeichenketten behandelt. Im Ausdrucksmodus werden Zahlen und Operationen verarbeitet. Als Faustregel gilt: Wenn eine Zeile mit einem Buchstaben oder den Sonderzeichen kaufmännisches Und [&], Punkt [.] oder Schrägstrich [\] beginnt, dann ist die Zeile im Befehlsmodus. Wenn die Zeile mit einer Zahl, einem Anführungszeichen (["] oder ['']), einer runden Klammer [(] oder dem [@]-Zeichen („Klammeraffe“) beginnt, dann ist die Zeile im Ausdrucksmodus.

Befehls- und Ausdrucksmodus können gemischt werden. Dabei muss man in der Regel runde Klammern zur Abgrenzung verwenden. In einen Befehl kann ein Ausdruck durch Klammern eingebaut werden. Außerdem kann eine Pipeline mit einem Ausdruck beginnen. Die folgende Tabelle zeigt verschiedene Beispiele zur Erläuterung. `Echo` ist der Alias für `Write-Output`.

**Tabelle 3.3** Ausdrücke in der PowerShell

| Beispiel | Bedeutung |
|---|---|
| `2+3` | Ein Ausdruck – die PowerShell führt die Berechnung aus und liefert 5. |
| `echo 2+3` | Ein reiner Befehl. „2+3“ wird als Zeichenkette angesehen und ohne Auswertung auf dem Bildschirm ausgegeben. |
| `echo (2+3)` | Ein Befehl mit integriertem Ausdruck. Auf dem Bildschirm erscheint 5. |
| `2+3 \| echo` | Eine Pipeline, die mit einem Ausdruck beginnt. Auf dem Bildschirm erscheint 5. |
| `echo 2+3 \| 7+6` | Eine unerlaubte Eingabe. Ausdrücke dürfen in der Pipeline nur als erstes Element auftauchen. |
| `$a = Get-Process` | Ein Ausdruck mit integriertem Befehl. Das Ergebnis wird einer Variablen zugewiesen. |

| Beispiel | Bedeutung |
|---|---|
| `$a \| Get-Process` | Eine Pipeline, die mit einem Ausdruck beginnt. Der Inhalt von `$a` wird als Parameter an `Get-Process` übergeben. |
| `Get-Process \| $a` | Eine unerlaubte Eingabe. Ausdrücke dürfen in der Pipeline nur als erstes Element auftauchen. |
| „Anzahl der laufenden Prozesse: (Get-Process).Count“ | Es ist wohl nicht das, was gewünscht ist, denn die Ausgabe ist: Anzahl der laufenden Prozesse: `(Get-Process).Count` |
| „Anzahl der laufenden Prozesse: $((Get-Process).Count)“ | Jetzt ist die Ausgabe „Anzahl der laufenden Prozesse: 95“, weil `$( ... )` einen Unterausdruck (Subexpression) einleitet und dafür sorgt, dass `Get-Process` ausgeführt wird. |

## 3.4 Externe Befehle

Alle Eingaben, die nicht als Commandlets oder mathematische Formeln erkannt werden, werden als externe Anwendungen behandelt. Es können sowohl klassische Kommandozeilenbefehle (wie *ping.exe*, *ipconfig.exe* und *netstat.exe*) als auch Windows-Anwendungen ausgeführt werden.

Die Eingabe `c:\Windows\Notepad.exe` ist daher möglich, um den „beliebten“ Windows-Editor zu starten. Auf gleiche Weise können auch WSH-Skripte aus der PowerShell heraus gestartet werden.

Die folgende Bildschirmabbildung zeigt den Aufruf von *netstat.exe*. Zuerst wird die Ausgabe nicht gefiltert. Im zweiten Beispiel kommt zusätzlich das Commandlet `Select-String` zum Einsatz, das nur die Zeilen ausgibt, die das Wort „LDAP“ enthalten.

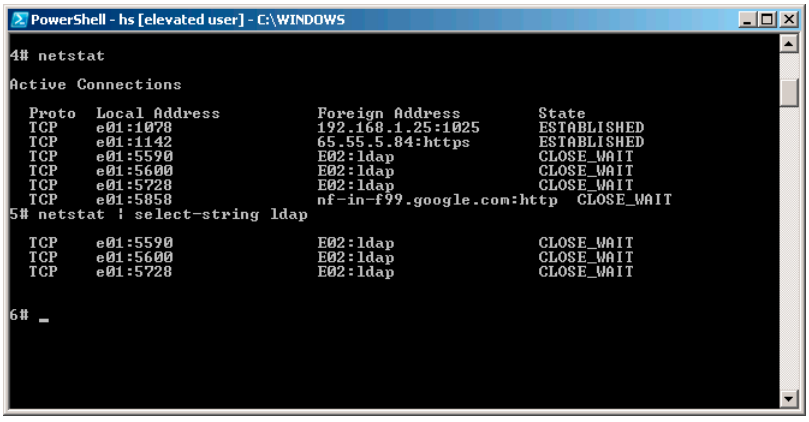



**Bild 3.14** Ausführung von netstat

Wenn ein Leerzeichen im Pfad zu einer .exe-Datei vorkommt, dann kann man die Datei so nicht aufrufen (hier wird nach einem Befehl „T:\data\software\Windows“ gesucht):

```
T:\data\software\Windows Tools\ImageEditor.exe
```

Auch die naheliegende Lösung der Verwendung von Anführungszeichen funktioniert nicht (hier wird die Zeichenkette ausgegeben):

```
"T:\data\software\Windows Tools\ImageEditor.exe"
```

Korrekt ist die Verwendung des kaufmännischen Und (&), das dafür sorgt, dass der Inhalt der Zeichenkette als Befehl betrachtet und ausgeführt wird:

```
& "T:\data\software\Windows Tools\ImageEditor.exe"
```

**ACHTUNG:** Grundsätzlich könnte es passieren, dass ein interner Befehl der PowerShell (Commandlet, Alias oder Function) genauso heißt wie ein externer Befehl. Die PowerShell warnt in einem solchen Fall nicht vor der Doppeldeutigkeit, sondern die Ausführung erfolgt nach folgender Präferenzliste:

- Aliase
- Funktionen
- Commandlets
- Externe Befehle

## 3.5 Dateinamen

Beim direkten Aufruf von Datendateien (z. B. `.doc`-Dateien) wird entsprechend den Windows-Einstellungen in der Registrierungsdatenbank die Standardanwendung gestartet und damit das Dokument geladen.

**HINWEIS:** Dateinamen und Ordnerpfade müssen nur in Anführungszeichen (einfache oder doppelte) gesetzt werden, wenn sie Leerzeichen enthalten.

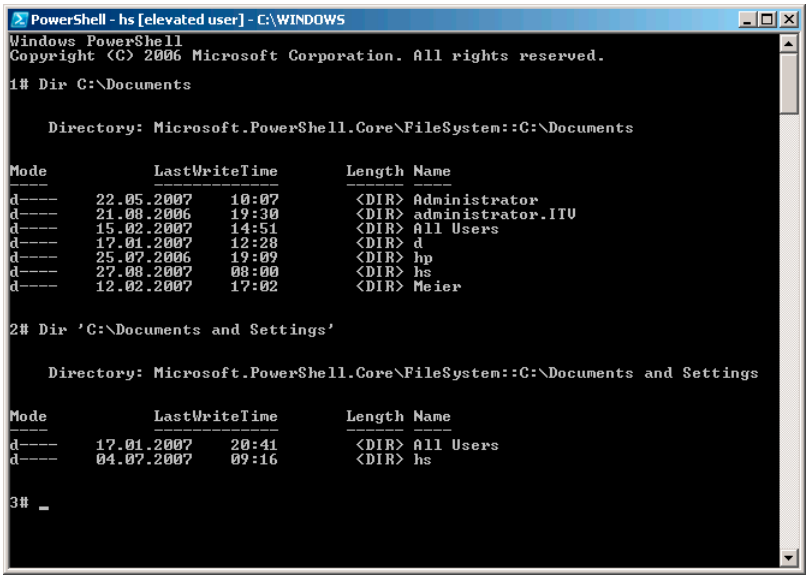



**Bild 3.15** Anführungszeichen bei Pfadangaben

# 4 Hilfefunktionen

Dieses Kapitel beschreibt die Hilfefunktionen der PowerShell.

## 4.1 Auflisten der verfügbaren Befehle

Die Liste aller verfügbaren Befehle (PowerShell-Commandlets, PowerShell-Funktionen, Aliase und klassische ausführbare Dateien) erhält man in der PowerShell auch durch

```
Get-Command
```

Dabei sind auch Muster erlaubt.

- `Get-Command Get-*` liefert alle Befehle, die mit „get“ anfangen.
- `Get-Command [gs]et-*` liefert alle Befehle, die mit „get“ oder „set“ anfangen.
- `Get-Command *-Service` liefert alle Befehle, die das Substantiv „Service“ besitzen.
- `Get-Command -noun Service` liefert ebenfalls alle Befehle, die das Substantiv „Service“ besitzen.
- `Get-Command *wmi*` liefert alle Befehle, die die Buchstabenfolge „wmi“ enthalten (und mutmaßlich mit der Windows Management Instrumentation zu tun haben).
- `Get-Command | Where-Object { $_.name -like "*cim*" -or $_.name -like "*wmi*" }` liefert alle Befehle, die die Buchstabenfolge „wmi“ oder „cmi“ enthalten. Ohne ein weiteres Commandlet `Where-Object`, das erst im nächsten Kapitel näher erläutert wird, ist diese Abfrage nicht machbar.

Das Commandlet `Get-Command` kann auch verwendet werden, um die Information zu erhalten, was die PowerShell unter einem Befehl versteht. `Get-Command` sucht nach angegebenen Namen in Commandlets, Aliasen, Funktionen, Skriptdateien und ausführbaren Dateien (siehe Bild 4.1).

Gibt man nach `Get-Command` den Namen einer *.exe*-Datei an, zeigt die PowerShell, in welchem Pfad die ausführbare Datei gefunden werden kann. Gesucht wird dabei nur in den Pfaden gemäß der Umgebungsvariablen `%Path%`.

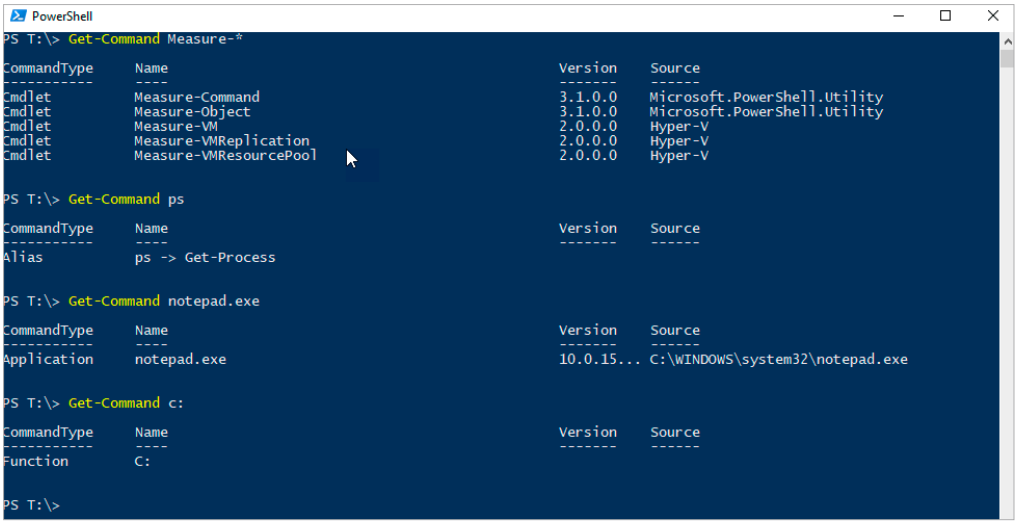



**Bild 4.1** Beispiele zum Einsatz von Get-Command

```
Get-Command *.exe
```

zeigt eine Liste aller direkt aufrufbaren ausführbaren Dateien.

Windows 10 (Stand Creators Update, alias Redstone 2, Versionsnummer 1703) mit Power Shell 5.1 bietet 1537 Commandlets. Die rasante Fortentwicklung der Funktionalität der PowerShell, aber auch die gravierende Abhängigkeit ihrer Mächtigkeit von dem installierten Betriebssystem, zeigt die folgende Tabelle.

**Tabelle 4.1** Wachstum der Mächtigkeit der Windows PowerShell

| PowerShell-Version | Betriebssystem | Anzahl der Commandlets und Funktionen |
|---|---|---|
| PowerShell 5.1 | Windows 10 (Redstone 2, „Creators Update“ vom 05.04.2017) | 1537 |
| PowerShell 5.0 | Windows 10 (Threshold 1, Ursprungsversion vom 29.05.2016) | 1404 |
| PowerShell 4.0 | Windows Server 2012 R2 | 1376 |
| PowerShell 4.0 | Windows 8.1 | 1132 |
| PowerShell 4.0 | Windows 7 | 573 |
| PowerShell 3.0 | Windows 8 | 945 |
| PowerShell 3.0 | Windows 7 | 561 |
| PowerShell 2.0 | Windows 7 | 273 |
| PowerShell 1.0 | Alle | 163 |

Ermitteln kann man diese Zahlen mit:

```
(Get-Command) | group commandtype
```

`Get-Command` liefert unter PowerShell seit 2.0 sowohl Commandlets als auch eingebaute Funktionen (deren Handhabung oft der von Commandlets entspricht, nur die Art der Implementierung ist anders). Unter PowerShell 1.0 musste man die Funktionen separat zählen mit:

```
(dir function:).count
```

Wenn Sie wissen möchten,

- welche Commandlets zwischen zwei Versionen hinzugekommen sind oder
- hinsichtlich welcher Commandlets sich zwei Systeme unterscheiden,

können Sie dies wie folgt ermitteln:

Auf dem einen System exportieren Sie eine Liste der Commandlets in eine Textdatei.

Auf einem System mit PowerShell 3.0 führen Sie folgende Befehle aus, um Commandlets und Funktionen zu exportieren:

```
Get-Command | ft name -hide | out-file h:\wps3_commandlets.txt
dir function: | ft Name -hide | out-file h:\wps3_commandlets.txt -Append
```

Auf einem System mit PowerShell ab Version 4.0 brauchen Sie nur einen Befehl (dieser exportiert Commandlets und Funktionen):

```
Get-Command | ft name -hide | out-file h:\wps\wps4_commandlets.txt
```

Dann führt man beide Textdateien auf einem System zusammen und führt dort aus:

```
$wps1 = Get-content H:\wps3_Commandlets.txt | sort
$wps2 = Get-content H:\wps4_Commandlets.txt | sort
compare-object $wps2 $wps4  -syncwindow 2000 | foreach {
[string]$_.Inputobject).Trim() } | out-file h:\wps4_Commandlets_neu.txt
```

## 4.2 Volltextsuche

Get-Command sucht nur in den Commandletnamen. Mit Get-Help kann man unter Angabe einer beliebigen Zeichenkette in den Hilfedateien suchen.

**Beispiel:** Get-Help „Local user account“

```
PS T:\> Get-help "Local user account"

Name                                Category  Module                    Synopsis
----                                --------  ------                    --------
Connect-PSSession                   Cmdlet    Microsoft.PowerShell.Core Reconnects to disconnected sessions.
Enter-PSSession                     Cmdlet    Microsoft.PowerShell.Core Starts an interactive session with a remote co...
Get-PSSession                       Cmdlet    Microsoft.PowerShell.Core Gets the Windows PowerShell sessions on local ...
Invoke-Command                      Cmdlet    Microsoft.PowerShell.Core Runs commands on local and remote computers.
New-PSSession                       Cmdlet    Microsoft.PowerShell.Core Creates a persistent connection to a local or ...
Receive-PSSession                   Cmdlet    Microsoft.PowerShell.Core Gets results of commands in disconnected sessi...
Invoke-RestMethod                   Cmdlet    Microsoft.PowerShell.U... Sends an HTTP or HTTPS request to a RESTful we...
Invoke-WebRequest                   Cmdlet    Microsoft.PowerShell.U... Gets content from a web page on the Internet.
Add-LocalGroupMember                Cmdlet    Microsoft.PowerShell.L... Adds members to a local group.
Disable-LocalUser                   Cmdlet    Microsoft.PowerShell.L... Disables a local user account.
Enable-LocalUser                    Cmdlet    Microsoft.PowerShell.L... Enables a local user account.
Get-LocalUser                       Cmdlet    Microsoft.PowerShell.L... Gets local user accounts.
New-LocalUser                       Cmdlet    Microsoft.PowerShell.L... Creates a local user account.
Remove-LocalGroupMember             Cmdlet    Microsoft.PowerShell.L... Removes members from a local group.
Remove-LocalUser                    Cmdlet    Microsoft.PowerShell.L... Deletes local user accounts.
Rename-LocalUser                    Cmdlet    Microsoft.PowerShell.L... Renames a local user account.
Set-LocalUser                       Cmdlet    Microsoft.PowerShell.L... Modifies a local user account.
Set-AssignedAccess                  Function  AssignedAccess            Configures a user to launch only one app.
about_ActivityCommonParameters      HelpFile                            Describes the parameters that Windows PowerShell
about_WorkflowCommonParameters      HelpFile                            This topic describes the parameters that are v...
about_ActivityCommonParameters      HelpFile                            Describes the parameters that Windows PowerShell
about_WorkflowCommonParameters      HelpFile                            This topic describes the parameters that are v...


PS T:\>
```

**Bild 4.2** Volltextsuche mit Get-Help

## 4.3 Erläuterungen zu den Befehlen

Einen Hilfetext zu einem Commandlet bekommt man über `Get-Help commandletname`, z. B.:

```
Get-Help Get-Process
```

Dabei kann man durch die Parameter `-detailed`, `-example` und `-full` mehr Hilfe erhalten.

Die Hilfe erscheint abhängig von der installierten Sprachversion der PowerShell. Der Autor dieses Buchs verwendet jedoch primär englische Betriebssysteme und Anwendungen.

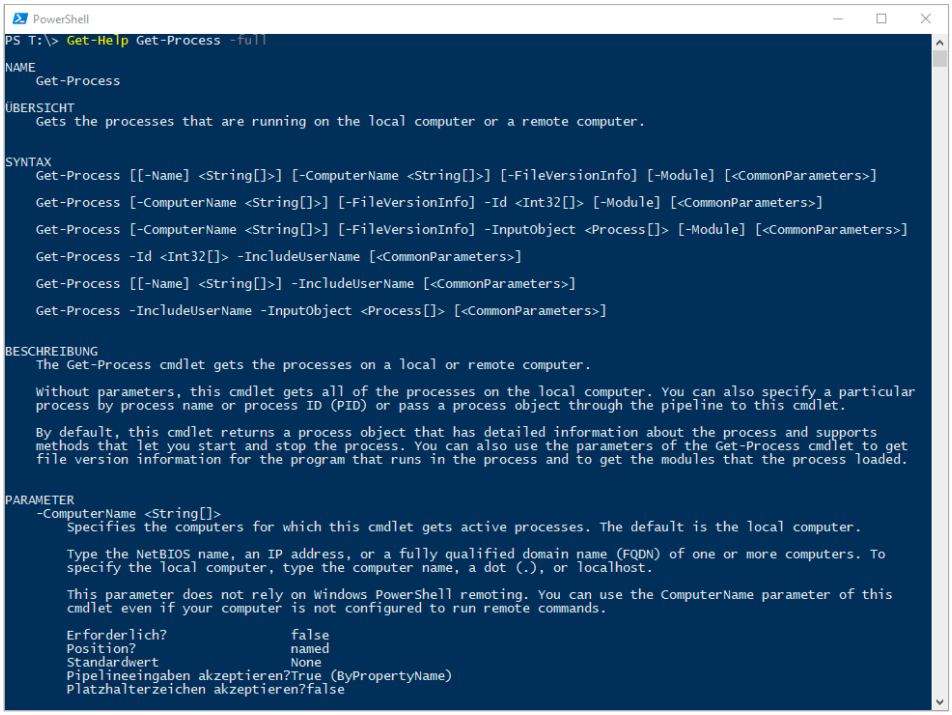



**Bild 4.3** Ausschnitt aus dem Hilfetext zum Commandlet Get-Process

**TIPP:** Alternativ zum Aufruf von `Get-Help` kann man auch den allgemeinen Parameter `-?` an das Commandlet anhängen, z. B. `Get-Process -?`. Dann erhält man die Kurzversion der Hilfe, hat aber keine Option für die ausführlicheren Versionen.

## 4.4 Hilfe zu Parametern

Um zu sehen, welche Parameter ein Befehl bietet, kann man `Get-Help` mit dem Parameter `-Parameter` verwenden:

```
Get-Help Get-Process -parameter "*" | ft name, type
```

Einige Commandlets (z. B. `New-Button` aus dem WPK (Windows Presentation Foundation (WPF) PowerShell Kit), siehe Kapitel 62 *„Grafische Benutzeroberflächen“*) haben sehr viele Parameter (in diesem Fall 180!). Hier kann man auch filtern:

```
Get-Help New-Button -parameter "on_*" | ft name, type
```

Genauere Hilfe zu einem einzelnen Parameter erhält man, wenn man nach `-parameter` den Namen angibt und die weitere Formatierung weglässt. Die folgende Bildschirmabbildung zeigt, wie man Hilfe zu dem Parameter `-Foregroundcolor` im Commandlet `Write-Host` erhält. Neben den möglichen Farbwerten sagt die Hilfe auch, dass

- die Angabe einer Farbe nicht erforderlich ist
- die Farbangabe nicht über die Position des Parameters gebunden wird, d. h., dass immer der Parametername anzugeben ist
- der Farbwert auch nicht aus der Pipeline eingelesen werden kann
- im Farbwert keine Platzhalter erlaubt sind

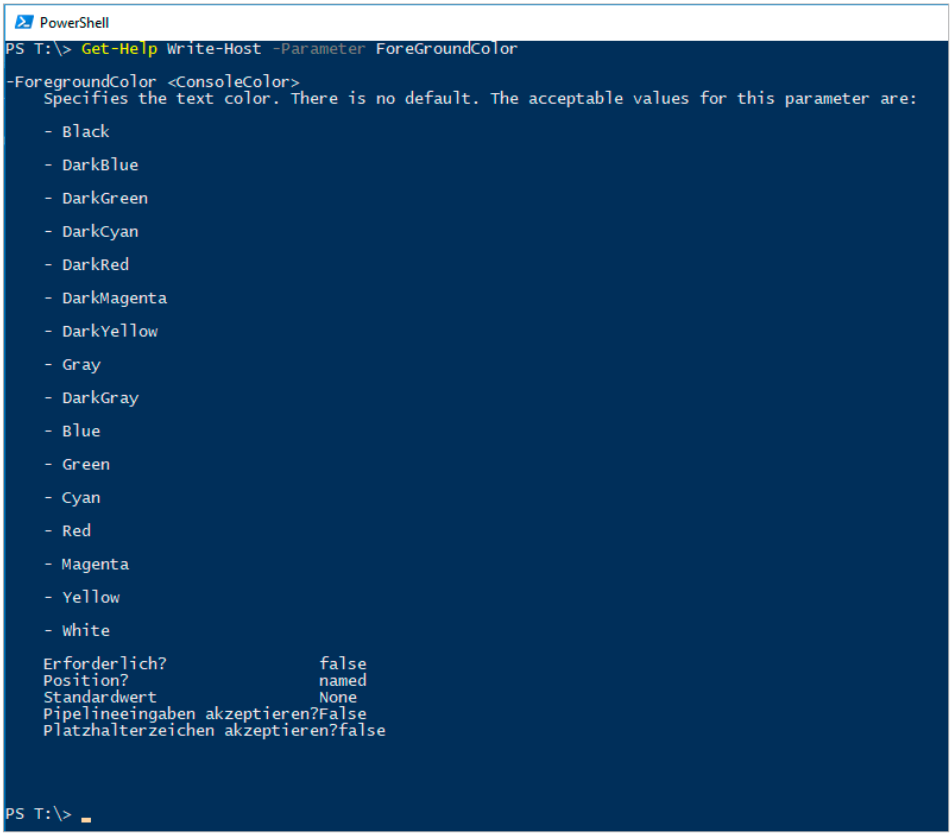



**Bild 4.4** Hilfe zu dem Parameter -ForeGroundColor beim Commandlet Write-Host

Schaut man sich hingegen die Hilfe zum Parameter `-Name` beim Commandlet `Get-Service` an, sieht man zwar weniger Text, aber mehr Möglichkeiten:

- Es kann nicht nur eine feste Menge von Zeichenketten, sondern eine beliebige Zeichenkette übergeben werden. Dies zeigt der Typ `<string>` an.
- Genau genommen steht da `<string[]>`. Die eckigen Klammern bedeuten „Menge“, es kann also nicht nur eine Zeichenkette, sondern auch eine Menge von Zeichenketten über-

geben werden (Beispiel: Dienste, die mit dem Buchstaben a beginnen oder enden oder mit x beginnen oder enden: `Get-Service -name "a*","*a","x*","*x"`).

- Der Wert kann über seine Position (0 bedeutet: an erster Stelle) übergeben werden. Daher kann man `-name` weglassen, sofern man den Wert für den Parameter an erster Stelle übergibt: `Get-Service "a*","*a","x*","*x"`
- Der Werte (oder die Werte) für den Parameter `-name` kann auch als Wert aus der Pipeline gelesen werden. Möglich ist also `"a*" | Get-Service` oder `"a*","*a","x*","*x" | Get-Service`

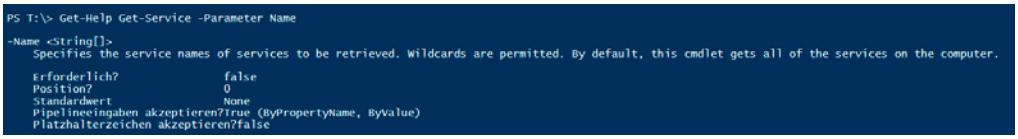
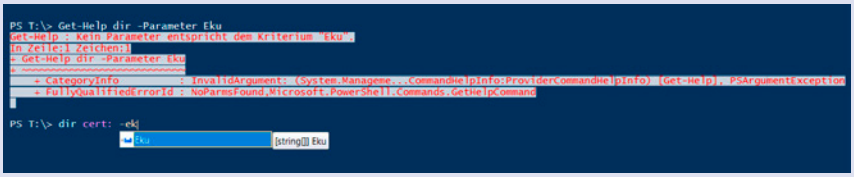



**Bild 4.5** Hilfe zu dem Parameter -Name beim Commandlet Get-Service

**HINWEIS:** Leider sind dynamische (d. h. von anderen Parametern abhängige) Parameter zu Commandlets nicht in der Hilfe verzeichnet.




## 4.5 Hilfe mit Show-Command

Die PowerShell ist kommandozeilenorientiert. Vor der PowerShell 3.0 gab es in der PowerShell nur zwei Befehle, die eine grafische Benutzeroberfläche zeigten: `Out-GridView` (zur Ausgabe von Objekten in einer filter- und sortierbaren Tabelle) und `Get-Credential` (zur Abfrage von Benutzername und Kennwort). Seit PowerShell 3.0 kann sich der PowerShell-Nutzer mit dem Commandlet `Show-Command` für jedes PowerShell-Commandlet und jede Function eine grafische Eingabemaske zeigen lassen.

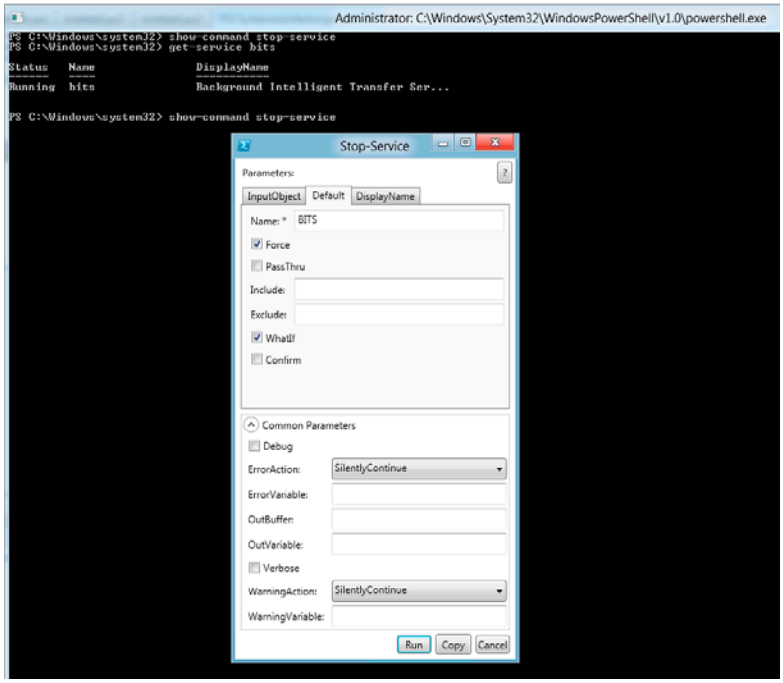



**Bild 4.6** Show-Command bietet Eingabehilfen für Einsteiger.

Das vorherige Bild zeigt dies für das Commandlet `Stop-Service`. Ziel von `Show-Command` ist es, insbesondere Einsteigern die Erfassung der Parameter zu erleichtern. Pflichtparameter sind mit einem Stern gekennzeichnet. Ein Klick auf die „Copy"-Schaltfläche legt den erzeugten Befehl in die Zwischenablage, ohne ihn auszuführen.

**TIPP:** Das Fenster „Befehls-Add-On" in dem ISE ist eine modifizierte Version von `Show-Command`.

## 4.6 Hilfefenster

Seit PowerShell 3.0 kann man auch aus der PowerShell-Konsole heraus ein eigenständiges Hilfefenster starten, indem man bei `Get-Help` den Parameter `-ShowWindow` verwendet.

```
Get-Help "Set-PrintConfiguration" -ShowWindow
```

Das Hilfefenster nutzt zur Hervorhebung fette Schrift, bietet eine Zoomfunktion und eine Volltextsuche an (vgl. Bild 4.7).

**HINWEIS:** Get-Help bietet in PowerShell Core keinen Parameter -ShowWindow mehr, da dieses Fenster auf der Windows Presentation Foundation (WPF) basiert, die es in .NET Core nicht gibt.

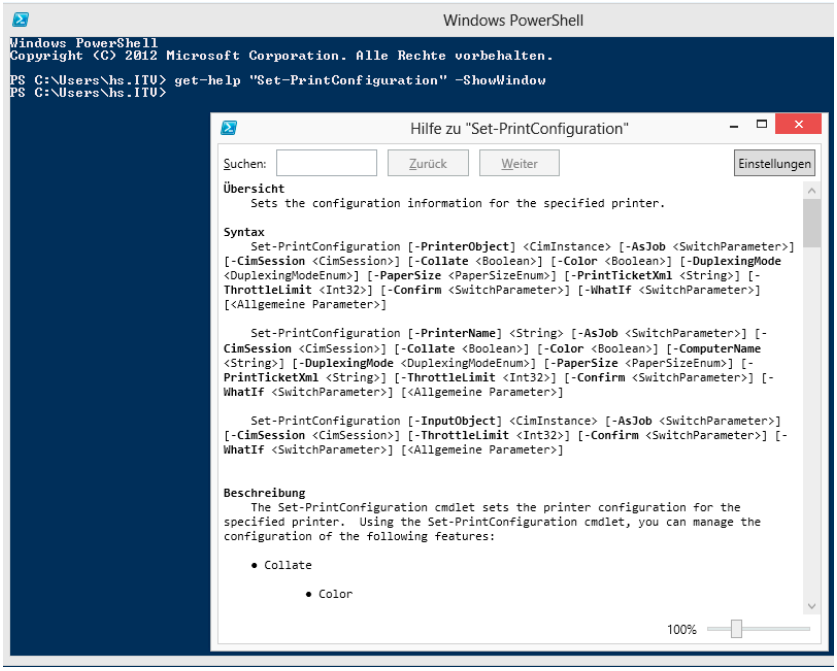



**Bild 4.7** Hilfefenster, das Get-Help durch den Parameter -ShowWindow startet

Eine grafische Hilfedatei im *.chm*-Dateiformat zur PowerShell gab es nur für die PowerShell 1.0 und 2.0 als Zusatz.

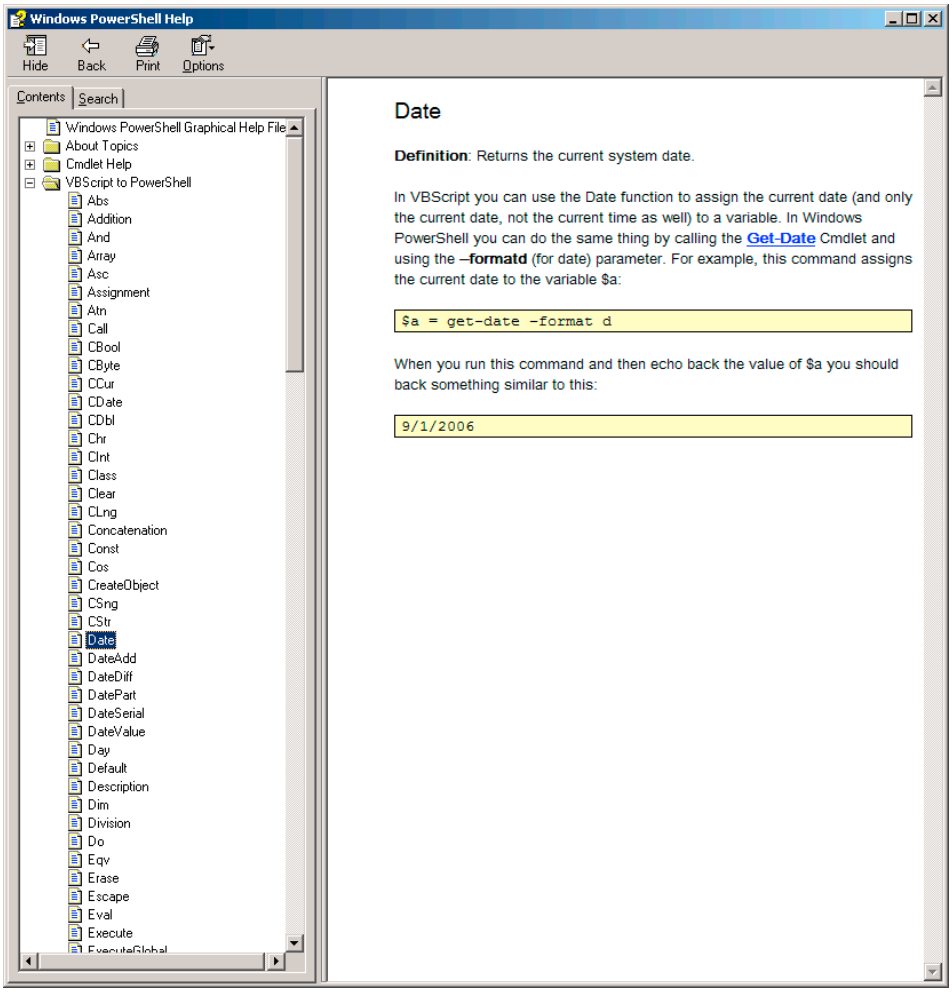



**Bild 4.8** Hilfe zum Transfer von VBScript nach PowerShell

## 4.7 Allgemeine Hilfetexte

Die PowerShell enthält auch einige allgemeine Hilfetexte. Diese Dokumente beginnen mit „about“. Man findet sie mit `Get-Help about_`. Ein konkretes Dokument ruft man dann unter Angabe des kompletten Dokumentennamen ab: z. B. `Get-Help about_arrays`.

```
PS T:\> Get-help about_

Name                               Category  Module                    Synopsis
----                               --------  ------                    --------
about_ActivityCommonParameters     HelpFile                            Describes the parameters that Windows PowerShell
about_Aliases                      HelpFile                            Describes how to use alternate names for cmdle...
about_Arithmetic_Operators         HelpFile                            Describes the operators that perform arithmeti...
about_Arrays                       HelpFile                            Describes arrays, which are data structures de...
about_Assignment_Operators         HelpFile                            Describes how to use operators to assign value...
about_Automatic_Variables          HelpFile                            Describes variables that store state informati...
about_Break                        HelpFile                            Describes a statement you can use to immediate...
about_Checkpoint-Workflow          HelpFile                            Describes the Checkpoint-Workflow activity, which
about_CimSession                   HelpFile                            Describes a CimSession object and the differen...
about_Classes                      HelpFile                            Describes how you can use classes to develop i...
about_Command_Precedence           HelpFile                            Describes how Windows PowerShell determines wh...
about_Command_Syntax               HelpFile                            Describes the syntax diagrams that are used in...
about_Comment_Based_Help           HelpFile                            Describes how to write comment-based help topi...
about_CommonParameters             HelpFile                            Describes the parameters that can be used with...
about_Comparison_Operators         HelpFile                            Describes the operators that compare values in...
about_Continue                     HelpFile                            Describes how the Continue statement immediate...
about_Core_Commands                HelpFile                            Lists the cmdlets that are designed for use wi...
about_Data_Sections                HelpFile                            Explains Data sections, which isolate text str...
about_Debuggers                    HelpFile                            Describes the Windows PowerShell debugger.
about_DesiredStateConfiguration    HelpFile                            Provides a brief introduction to the Windows
about_Do                           HelpFile                            Runs a statement list one or more times, subje...
about_Environment_Variables        HelpFile                            Describes how to access Windows environment va...
about_Escape_Characters            HelpFile                            Introduces the escape character in Windows Pow...
about_Eventlogs                    HelpFile                            Windows PowerShell creates a Windows event log...
about_Execution_Policies           HelpFile                            Describes the Windows PowerShell execution pol...
about_For                          HelpFile                            Describes a language command you can use to ru...
about_ForEach-Parallel             HelpFile                            Describes the ForEach -Parallel language const...
about_Foreach                      HelpFile                            Describes a language command you can use to tr...
about_Format.ps1xml                HelpFile                            The Format.ps1xml files in Windows PowerShell ...
about_Functions                    HelpFile                            Describes how to create and use functions in W...
about_Functions_Advanced           HelpFile                            Introduces advanced functions that act similar...
about_Functions_Advanced_Methods   HelpFile                            Describes how functions that specify the Cmdle...
about_Functions_Advanced_Param...  HelpFile                            Explains how to add parameters to advanced fun...
about_Functions_CmdletBindingA...  HelpFile                            Describes the attribute that makes a function ...
about_Functions_OutputTypeAttr...  HelpFile                            Describes an attribute that reports the type o...
```

**Bild 4.9** Ausschnitt aus der Liste der „About“-Dokumente

## 4.8 Aktualisieren der Hilfedateien

Die Hilfeinformationen, die durch `Get-Help` ausgelesen werden können, sind in XML-Dateien gespeichert. Das verwendete XML-Format heißt Microsoft Assistance Markup Language (MAML). Die Hilfe-Dateien sind den einzelnen Modulen zugeordnet.

```
Microsoft.PowerShe...ement.dll-help.xml*
9897    <command:syntaxItem>
9898     <maml:name>Get-Process</maml:name>
9899     <command:parameter required="false" variableLength="true" globbing="true" pipelineInput="true (ByPropertyName)"
position="1" aliases="">
9900      <maml:name>Name</maml:name>
9901      <maml:description>
9902       <maml:para>Specifies one or more processes by process name. You can type multiple process names (separated by commas)
and use wildcard characters. The parameter name ("Name") is optional.</maml:para>
9903      </maml:description>
9904      <command:parameterValue required="true" variableLength="true">String[]</command:parameterValue>
9905     </command:parameter>
9906     <command:parameter required="false" variableLength="true" globbing="false" pipelineInput="true (ByPropertyName)"
position="named" aliases="">
9907      <maml:name>ComputerName</maml:name>
9908      <maml:description>
9909       <maml:para>Gets the processes running on the specified computers. The default is the local computer.</maml:para>
9910       <maml:para>Type the NetBIOS name, an IP address, or a fully qualified domain name of one or more computers. To specify
 the local computer, type the computer name, a dot (.), or "localhost".</maml:para>
9911       <maml:para>This parameter does not rely on Windows PowerShell remoting. You can use the ComputerName parameter of Get-
Process even if your computer is not configured to run remote commands.</maml:para>
9912      </maml:description>
9913      <command:parameterValue required="true" variableLength="true">String[]</command:parameterValue>
9914     </command:parameter>
9915     <command:parameter required="false" variableLength="false" globbing="false" pipelineInput="false" position="named"
aliases="">
9916      <maml:name>FileVersionInfo</maml:name>
9917      <maml:description>
9918       <maml:para>Gets the file version information for the program that runs in the process. </maml:para>
9919       <maml:para>On Windows Vista and later versions of Windows, you must open Windows PowerShell with the "Run as
administrator" option to use this parameter on processes that you do not own. </maml:para>
9920       <maml:para>You cannot use the FileVersionInfo and ComputerName parameters of the Get-Process cmdlet in the same
command. To get file version information for a process on a remote computer, use the Invoke-Command cmdlet.</maml:para>
```

**Bild 4.10** Ausschnitt aus der Hilfedatei Microsoft.PowerShell.Commands.Management.dll-help.xml

Mit PowerShell 3.0 hatte Microsoft die Möglichkeit eingeführt, die Hilfe-Dateien aus der laufenden PowerShell heraus zu aktualisieren („Updatable Help System“). Die Ausführung des Commandlets `Update-Help` kontaktiert den Microsoft-Downloadserver (download.microsoft.com) und aktualisiert im laufenden Betrieb die Hilfedateien. Auch wenn es sich um relativ kleine Dateien handelt (aktuell insgesamt nur rund 10 MB), dauert der Download über eine 50-MBit-Leitung zwei bis drei Minuten. Der Download besteht für jedes PowerShell-Modul aus einer sogenannten Help-Info-Datei, die als wesentliche Information die Sprache und die Versionsnummer enthält, sowie einer komprimierten Datei (ZIP-Format, Dateinamenserweiterung ist aber CAB), die nur heruntergeladen wird, wenn die lokalen Hilfeinformationen nicht auf dem aktuellen Stand sind.

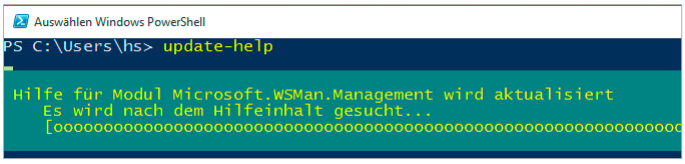



**Bild 4.11**
Aktualisieren der Hilfe mit Update-Help

**HINWEIS:** Die Aktualisierung der Hilfedateien für alle Standardmodule, die sich im c:\Windows\System32\WindowsPowerShell-Verzeichnis befinden, ist nur mit administrativen Rechten möglich.

**Listing 4.1** Beispiel für eine Help-Info-Datei

```
<?xml version="1.0" encoding="utf-8"?>
<HelpInfo xmlns="http://schemas.microsoft.com/powershell/help/2010/05">
<HelpContentURI>http://go.microsoft.com/fwlink/?linkid=210601</HelpContentURI>
  <SupportedUICultures>
    <UICulture>
      <UICultureName>en-US</UICultureName>
      <UICultureVersion>3.1.0.0</UICultureVersion>
    </UICulture>
  </SupportedUICultures>
</HelpInfo>
```

`Update-Help` kann durch Angabe eines Modulnamens im Parameter `-Module` die Hilfe für ein einzelnes Modul aktualisieren.

`Update-Help` kann durch Angabe eines Pfads im Parameter `-SourcePath` die Hilfedateien von einem lokalen Dateisystempfad oder Netzwerkpfad laden. Zu diesem Zweck kann man mit `Save-Help` die Help-Info-Dateien und die CAB-Dateien herunterladen. Größere Unternehmen können so die Hilfedateien zentral für alle Nutzer im Unternehmensnetzwerk bereitstellen.

**TIPP:** Die Aktualisierung der Hilfedateien kann auch im Editor „ISE“ im Menü „Hilfe“ ausgelöst werden.

## 4.9 Online-Hilfe

Die Dokumentation der PowerShell findet man hier:

*http://technet.microsoft.com/en-us/library/bb978526.aspx*

Die zusätzlichen betriebssystemabhängigen PowerShell-Module sind hier dokumentiert:

*https://technet.microsoft.com/library/dn249523.aspx*

Sie werden aber feststellen, dass dort jedes Commandlet einzeln beschrieben ist. Es gibt aber leider keine Dokumente, die das komplexere Zusammenspiel von Commandlets erklären oder die Vorgehensweise anhand von Praxisgebieten beschreiben wie in diesem Buch.

**HINWEIS:** Neu seit PowerShell 3.0 ist der Parameter `-Online` beim Commandlet `Get-Help`, der für ein Commandlet direkt die passende Seite in der Online-Hilfe öffnet.

Eine Online-Hilfe des Buchautors ist die Website *www.dotnet-lexikon.de*, wo Sie zu vielen Begriffen rund um PowerShell und .NET Erklärungstexte sowie ein Abkürzungsverzeichnis finden.

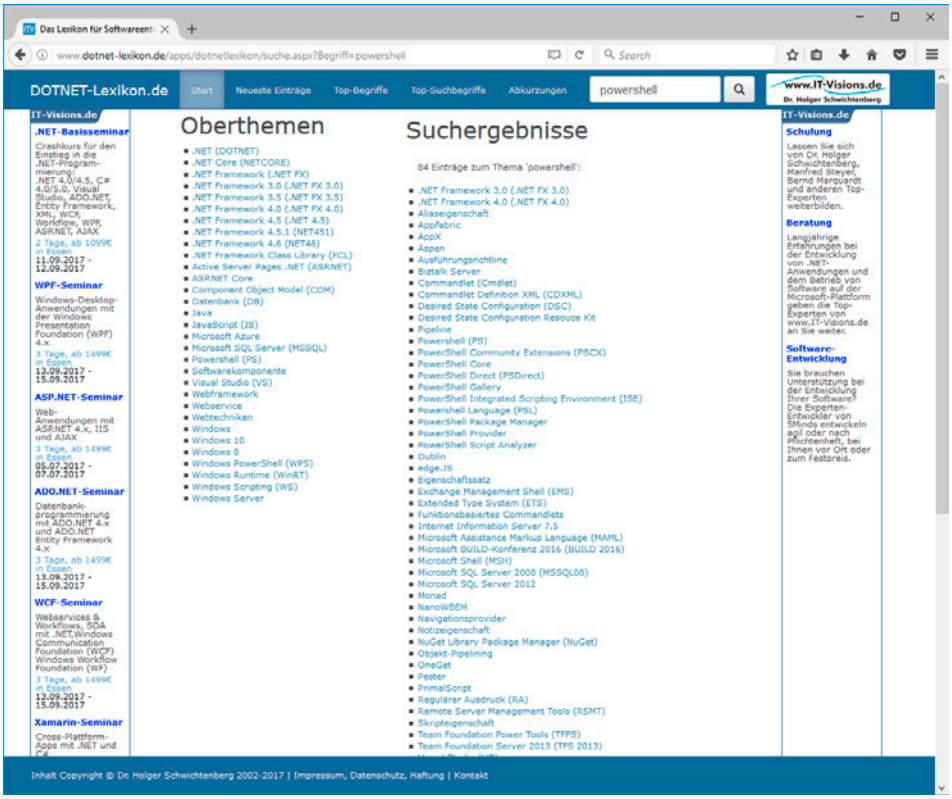



**Bild 4.12** Hilfe zu den Fachbegriffen und Abkürzungen auf der Website *www.dotnet-lexikon.de*

## 4.10 Fehlende Hilfetexte

Leider gibt es nicht zu allen Commandlets eine Hilfe. Microsoft wird in seinem Softwareentwicklungsprozess immer agiler und vernachlässigt dabei leider die Dokumentation. So gibt es zum Beispiel zu einigen in PowerShell 5.1 eingeführten Commandlets wie `Test-FileCatalog` zum Redaktionsschluss dieses Buchs immer noch keine adäquaten Hilfetexte, auch wenn PowerShell 5.1 schon vor einigen Monaten erschienen ist. So zeigt `Get-Help` hier genau wie die Webseite nur die im Commandlet automatisch verfügbaren Metadaten über die Parameter, aber keinerlei Erläuterungstexte und keine Beispiele.

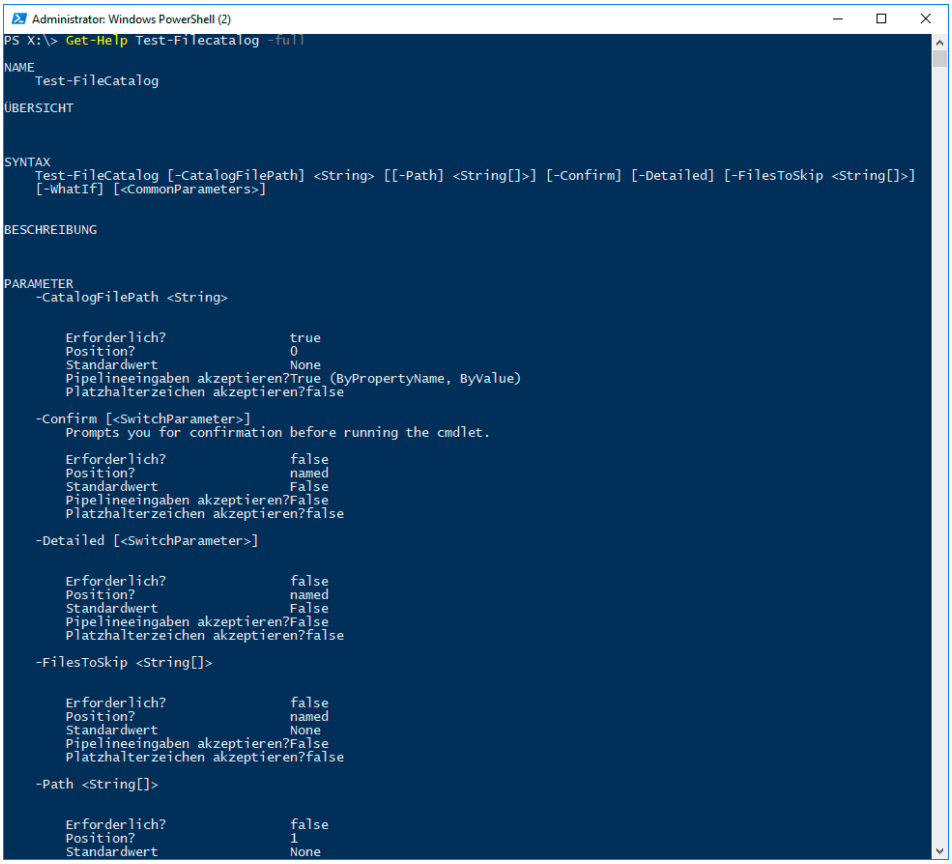



**Bild 4.13** Keine Hilfetexte zum Commandlet und zu den Parametern bei Test-Filecatalog

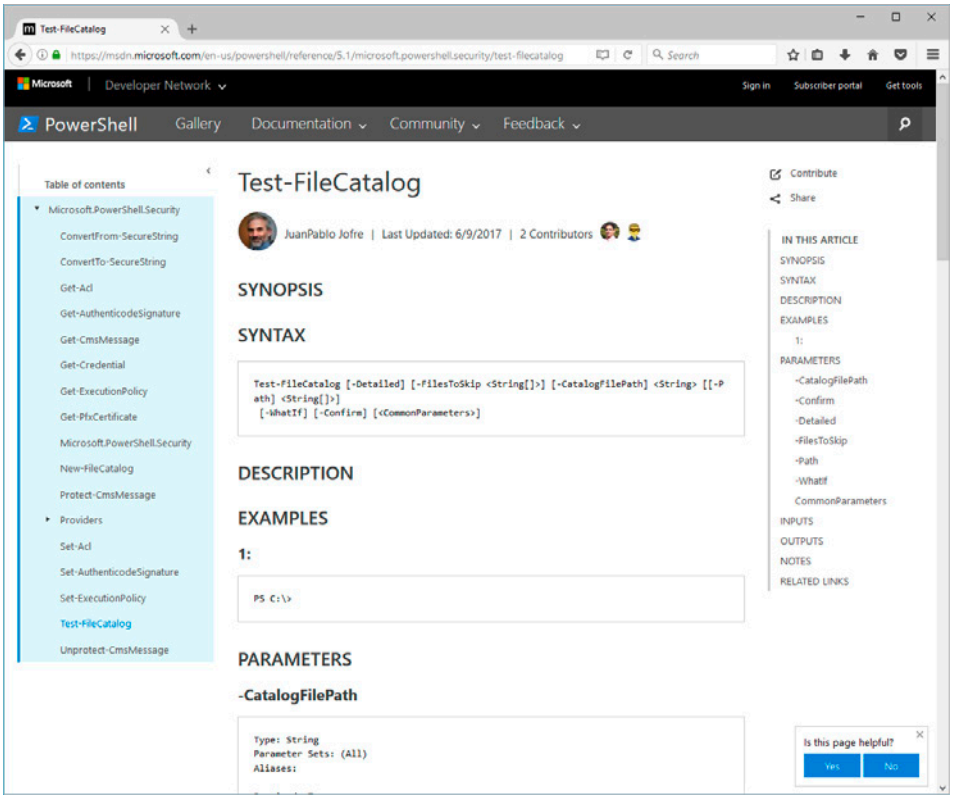



**Bild 4.14** Auch auf der Website *[https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.security/test-filecatalog]* gibt es keine Hilfetexte zum Commandlet und zu den Parametern bei Test-Filecatalog.

## 4.11 Dokumentation der .NET-Klassen

Informationen zu den .NET-Klassen, mit denen die PowerShell arbeitet, finden Sie an folgenden Stellen:

- PowerShell-Dokumentation für den Namensraum `System.Management.Automation`
- Dokumentation der .NET-Framework-Klassenbibliothek in der Microsoft Developer Network Library (MSDN Library). Diese gibt es offline in Verbindung mit Microsoft Visual Studio oder online unter *http://msdn.microsoft.com/en-us/library/gg145045.*aspx.
- Produktspezifische Dokumentationen, z. B. Exchange-Server-Dokumentation oder System-Center-Dokumentation

Die Dokumentation zeigt die verfügbaren Klassenmitglieder (Attribut, Methoden, Ereignisse, Konstruktoren) (siehe Bild 4.15).

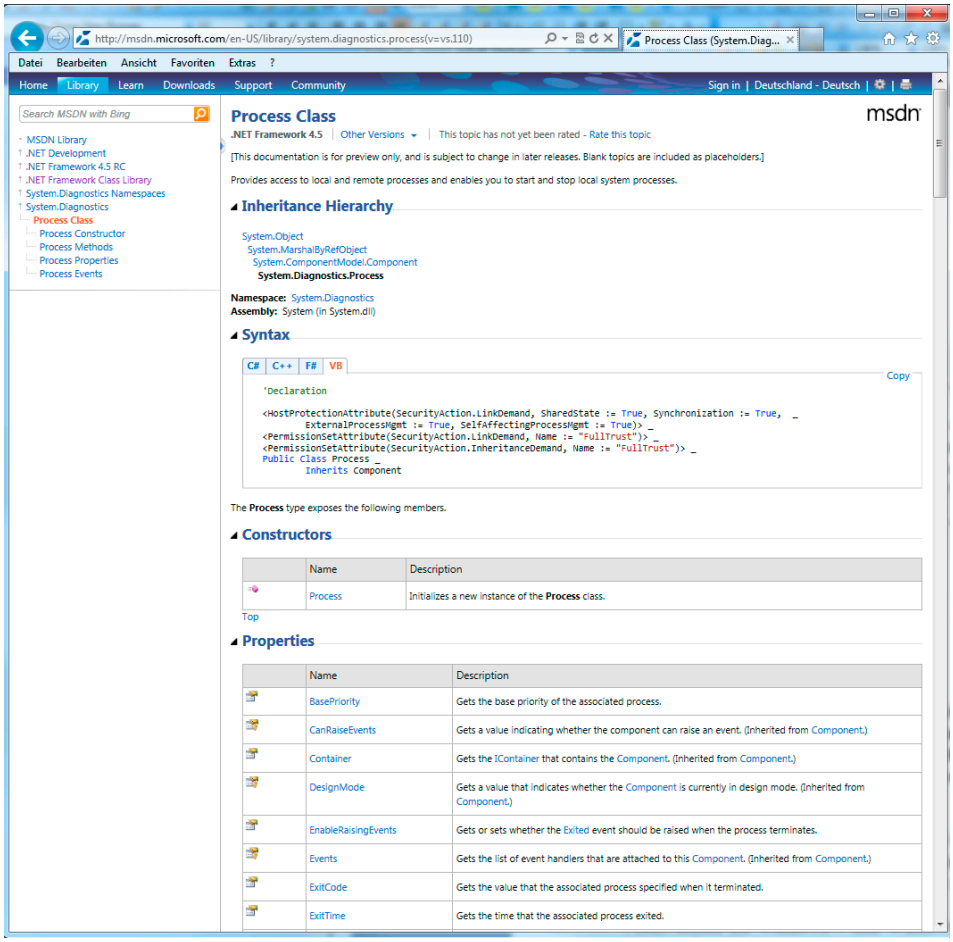



**Bild 4.15** Ausschnitt aus der Dokumentation der .NET-Klasse System.Diagnostics.Process (hier in der Online-Variante)

Das folgende Bild zeigt die Dokumentation der Klasse `Process` im Namensraum `System.Diagnostics`. In dem Baum links erkennt man die verschiedenen Arten von Mitgliedern: *Methoden* (Methods), *Eigenschaften* (Properties) und *Ereignisse* (Events).

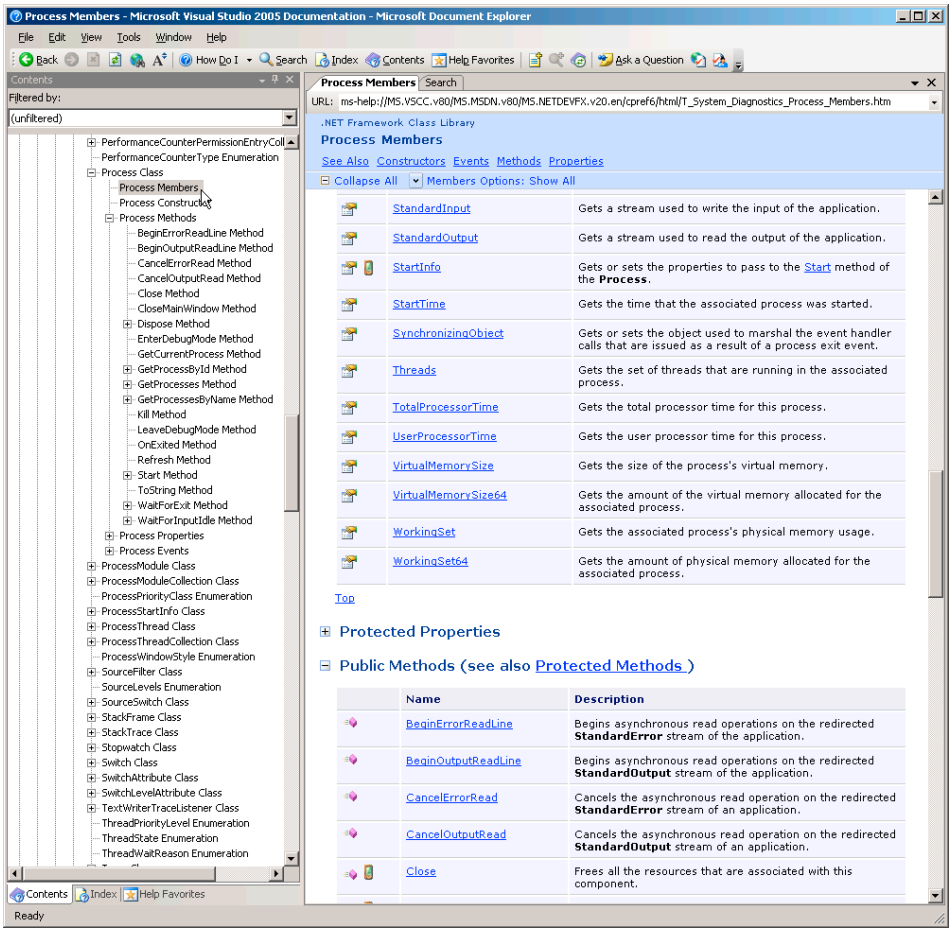



**Bild 4.16** Ausschnitt aus der Dokumentation der .NET-Klasse System.Diagnostics.Process (hier in der Offline-Variante, die mit Visual Studio mitgeliefert wird)

**HINWEIS:** Da die Dokumentation der .NET-Klassen für Softwareentwickler geschrieben wurde, ist sie häufig zu detailliert für PowerShell-Anwender. Leider ist derzeit noch keine für die Bedürfnisse von Administratoren angepasste Version absehbar.

**TIPP:** Die englische Dokumentation ist der deutschen vorzuziehen, weil es in den deutschen Übersetzungen viele Übersetzungsfehler gibt, die das Verständnis erschweren.

# 5 Objektorientiertes Pipelining

Ihre Mächtigkeit entfaltet die PowerShell erst durch das objektorientierte Pipelining, also durch die Weitergabe von strukturierten Daten von einem Commandlet zum anderen.

**HINWEIS:** Dieses Kapitel setzt ein Grundverständnis des Konzepts der Objektorientierung voraus. Wenn Sie diese Grundkenntnisse nicht besitzen, lesen Sie bitte zuvor im Anhang den Crashkurs „Objektorientierung“ sowie den Crashkurs „.NET Framework“ oder vertiefende Literatur.

## 5.1 Pipeline-Operator

Für eine Pipeline wird – wie auch in Unix-Shells üblich und in der normalen Windows-Konsole möglich – der vertikale Strich „|“ (genannt „Pipe“ oder „Pipeline Operator“) verwendet.

```
Get-Process | Format-List
```

bedeutet, dass das Ergebnis des `Get-Process`-Commandlets an `Format-List` weitergegeben werden soll. Die Standardausgabeform von `Get-Process` ist eine Tabelle. Durch `Format-List` werden die einzelnen Attribute der aufzulistenden Prozesse untereinander statt in Spalten ausgegeben.

Die Pipeline kann beliebig lang sein, d.h., die Anzahl der Commandlets in einer einzigen Pipeline ist nicht begrenzt. Man muss aber jedes Mal den Pipeline-Operator nutzen, um die Commandlets zu trennen.

Ein Beispiel für eine komplexere Pipeline lautet:

```
Get-ChildItem h:\daten –r -filter *.doc
| Where-Object { $_.Length -gt 40000 }
| Select-Object Name, Length
| Sort-Object Length
| Format-List
```

`Get-ChildItem` ermittelt alle Microsoft-Word-Dateien im Ordner *h:\Daten* und in seinen Unterordnern. Durch das zweite Commandlet (`Where-Object`) wird die Ergebnismenge auf diejenigen Objekte beschränkt, bei denen das Attribut `Length` größer ist als 40 000. `Select-Object` beschneidet alle Attribute aus `Name` und `Length`. Durch das vierte Commandlet in der Pipeline wird die Ausgabe nach dem Attribut `Length` sortiert. Das letzte Commandlet schließlich erzwingt eine Listendarstellung.

Nicht alle Aneinanderreihungen von Commandlets ergeben einen Sinn. Einige Aneinanderreihungen sind auch gar nicht erlaubt. Die Reihenfolge der einzelnen Befehle in der Pipeline ist nicht beliebig. Keineswegs kann man im obigen Befehl die Sortierung hinter die Formatierung setzen, weil nach dem Formatieren zwar noch ein Objekt existiert, dieses aber einen Textstrom repräsentiert. `Where-Object` und `Sort-Object` könnte man vertauschen; aus Gründen des Ressourcenverbrauchs sollte man aber erst einschränken und dann die verringerte Liste sortieren. Ein Commandlet kann aus vorgenannten Gründen erwarten, dass es bestimmte Arten von Eingabeobjekten gibt. Am besten sind aber Commandlets, die jede Art von Eingabeobjekt verarbeiten können.

Eine automatische Optimierung der Befehlsfolge wie in der Datenbankabfrage SQL gibt es bei PowerShell nicht.

Seit PowerShell-Version 3.0 hat Microsoft für den Zugriff auf das aktuelle Objekt der Pipeline zusätzlich zum Ausdruck `$_` den Ausdruck `$PSItem` eingeführt. `$_` und `$PSItem` sind synonym. Microsoft hat `$PSItem` eingeführt, weil einige Benutzer das Feedback gaben, dass `$_` zu (Zitat) „magisch“ sei.

**ACHTUNG:** Die PowerShell erlaubt beliebig lange Pipelines und es gibt auch Menschen, die sich einen Spaß daraus machen, möglichst viel durch eine einzige Befehlsfolge mit sehr vielen Pipes auszudrücken. Solche umfangreichen Befehlsfolgen sind aber meist für andere Menschen extrem schlecht lesbar. Bitte befolgen Sie daher den folgenden Ratschlag: Schreiben Sie nicht alles in eine einzige Befehlsfolge, nur weil es geht. Teilen Sie besser die Befehlsfolgen nach jeweils drei bis vier Pipe-Symbolen durch den Einsatz von Variablen auf (wird in diesem Kapitel auch beschrieben!) und lassen Sie diese geteilten Befehlsfolgen dann besser als PowerShell-Skripte ablaufen (siehe nächstes Kapitel).

## 5.2 .NET-Objekte in der Pipeline

Objektorientierung ist die herausragende Eigenschaft der PowerShell: Commandlets können durch Pipelines mit anderen Commandlets verbunden werden. Anders als Pipelines in Unix-Shells tauschen die Commandlets der PowerShell keine Zeichenketten, sondern typisierte .NET-Objekte aus. Das objektorientierte Pipelining ist im Gegensatz zum in den Unix-Shells und in der normalen Windows-Shell *(cmd.exe )* verwendeten zeichenkettenbasierten Pipelining nicht abhängig von der Position der Informationen in der Pipeline.

Ein Commandlet kann auf alle Attribute und Methoden der .NET-Objekte, die das vorhergehende Commandlet in die Pipeline gelegt hat, zugreifen. Die Mitglieder der Objekte können entweder durch Parameter der Commandlets (z. B. in `Sort-Object Length`) oder durch den expliziten Verweis auf das aktuelle Pipeline-Objekt (`$_`) in einer Schleife oder Bedingung (z. B. `Where-Object { $_.Length -gt 40000 }`) genutzt werden.

In einer Pipeline wie

```
Get-Process | Where-Object {$_.name -eq "iexplore"} | Format-Table ProcessName,
WorkingSet64
```

ist das dritte Commandlet daher nicht auf eine bestimmte Anordnung und Formatierung der Ausgabe von vorherigen Commandlets angewiesen, sondern es greift über den sogenannten Reflection-Mechanismus (den eingebauten Komponentenerforschungsmechanismus des .NET Frameworks) direkt auf die Eigenschaften der Objekte in der Pipeline zu.

**HINWEIS:** Genau genommen bezeichnet Microsoft das Verfahren als „Extended Reflection“ bzw. „Extended Type System (ETS)“, weil die PowerShell in der Lage ist, Objekte um zusätzliche Eigenschaften anzureichern, die in der Klassendefinition gar nicht existieren.

Im obigen Beispiel legt `Get-Process` ein .NET-Objekt der Klasse `System.Diagnostics.Process` für jeden laufenden Prozess in die Pipeline. `System.Diagnostics.Process` ist eine Klasse aus der .NET-Klassenbibliothek. Commandlets können aber jedes beliebige .NET-Objekt in die Pipeline legen, also auch einfache Zahlen oder Zeichenketten, da es in .NET keine Unterscheidung zwischen elementaren Datentypen und Klassen gibt. Eine Zeichenkette in die Pipeline zu legen, wird aber in der PowerShell die Ausnahme bleiben, denn der typisierte Zugriff auf Objekte ist wesentlich robuster gegenüber möglichen Änderungen als die Zeichenkettenauswertung mit regulären Ausdrücken.

Deutlicher wird der objektorientierte Ansatz, wenn man als Attribut keine Zeichenkette heranzieht, sondern eine Zahl. `WorkingSet64` ist ein 64 Bit langer Zahlenwert, der den aktuellen Speicherverbrauch eines Prozesses repräsentiert. Der folgende Befehl liefert alle Prozesse, die aktuell mehr als 20 Megabyte verbrauchen:

```
Get-Process | Where-Object {$_.WorkingSet64 -gt 20*1024*1024 }
```

Anstelle von `20*1024*1024` hätte man auch das Kürzel „20MB“ einsetzen können. Außerdem kann man `Where-Object` mit einem Fragezeichen abkürzen. Die kurze Variante des Befehls wäre dann also:

```
ps | ? {$_.ws -gt 20MB }
```

Wenn nur ein einziges Commandlet angegeben ist, dann wird das Ergebnis auf dem Bildschirm ausgegeben. Auch wenn mehrere Commandlets in einer Pipeline zusammengeschaltet sind, wird das Ergebnis des letzten Commandlets auf dem Bildschirm ausgegeben. Wenn das letzte Commandlet keine Daten in die Pipeline wirft, erfolgt keine Ausgabe.

## 5.3 Pipeline Processor

Für die Übergabe der .NET-Objekte zwischen den Commandlets sorgt der *PowerShell Pipeline Processor* (siehe folgende Grafik). Die Commandlets selbst müssen sich weder um die Objektweitergabe noch um die Parameterauswertung kümmern.

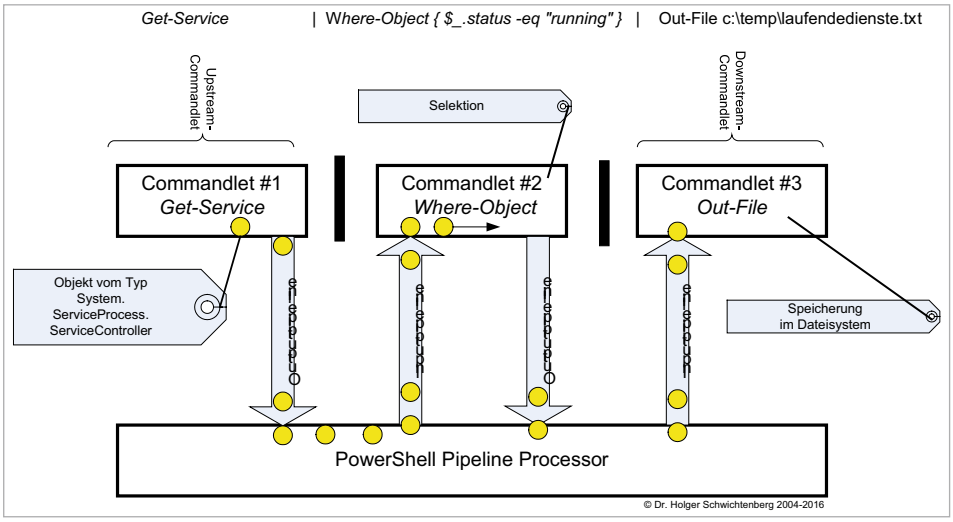



**Bild 5.1** Der Pipeline Processor befördert die Objekte vom Downstream-Commandlet zum Upstream-Commandlet. Die Verarbeitung ist in der Regel asynchron.

Wie das obige Bild schon zeigt, beginnt ein nachfolgendes Commandlet mit seiner Arbeit, sobald es ein erstes Objekt aus der Pipeline erhält. Das Objekt durchläuft die komplette Pipeline. Erst dann wird das nächste Objekt vom ersten Commandlet abgeholt. Man nennt dies „Streaming-Verarbeitung". Streaming-Verarbeitung ist schneller als die klassische sequentielle Verarbeitung, weil die folgenden Commandlets in der Pipeline nicht auf vorhergehende warten müssen. Intern arbeitet die PowerShell aber nur mit einem Thread, d. h. es findet keine parallele Verarbeitung mehrerer Befehle statt.

Aber nicht alle Commandlets beherrschen die asynchrone Streaming-Verarbeitung. Commandlets, die alle Objekte naturgemäß erst mal kennen müssen, bevor sie überhaupt ihren Zweck erfüllen können (z. B. `Sort-Object` zum Sortieren und `Group -Object` zum Gruppieren), blockieren die asynchrone Verarbeitung.

**HINWEIS:** Es gibt auch einige Commandlets, die zwar asynchron arbeiten könnten, aber leider nicht so programmiert wurden, um dies zu unterstützen.

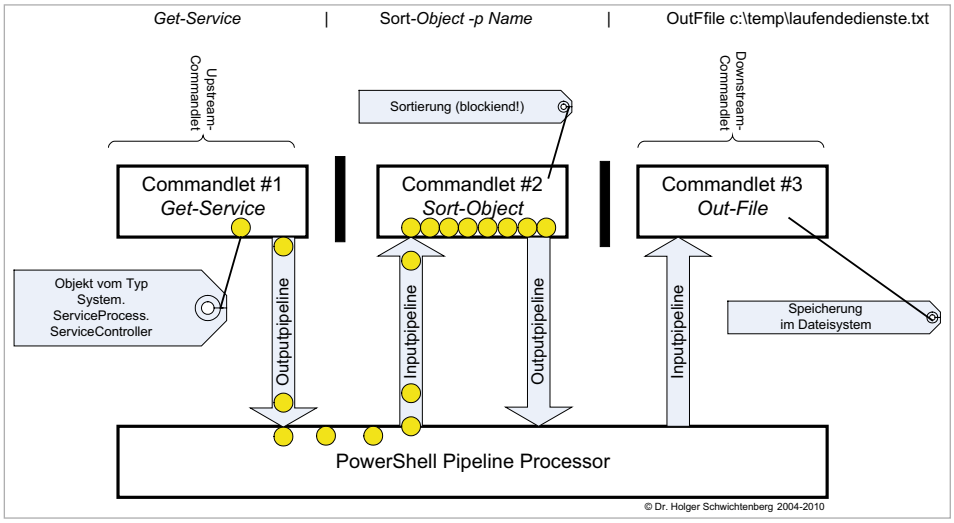



**Bild 5.2** Sort-Object blockiert die direkte Weitergabe. Erst wenn alle Objekte angekommen sind, kann das Commandlet sortieren.

Auch bei Commandlets, die Streaming-Verarbeitung unterstützen, kann der PowerShell-Nutzer mit dem allgemeinen Parameter `-OutBuffer` (abgekürzt `-ob`), den jedes Commandlet anbietet, dafür sorgen, dass eine bestimmte Anzahl von Objekten angesammelt wird, bevor eine Weitergabe an das nachfolgende Commandlet erfolgt.

Im Standard beginnt die Ausgabe der Ordner- und Dateinamen sofort:

```
dir c:\ -Recurse | ft name
```

In diesem Fall passiert lange nichts, bevor die Ausgabe beginnt:

```
dir c:\ -Recurse -OutBuffer:100000 | ft name
```

## 5.4 Pipelining von Parametern

Die Pipeline kann jegliche Art von Information befördern, auch einzelne elementare Daten. Einige Commandlets unterstützen es, dass auch die Parameter aus der Pipeline ausgelesen werden. Der folgende Pipeline-Befehl führt zu einer Auflistung aller Windows-Systemdienste, die mit dem Buchstaben „I“ beginnen.

```
"i*" | Get-Service
```

Die folgende Bildschirmabbildung zeigt einige Parameter des Commandlets `Get-Service`. Diese Liste erhält man durch den Befehl `Get-Help Get-Service -Parameter *`.

Interessant sind die mit gelbem Pfeil markierten Stellen. Nach „Accept pipeline Input“ kann man jeweils nachlesen, ob der Parameter des Commandlets aus den vorhergehenden Objekten in der Pipeline „befüttert“ werden kann.

Bei „`-Name`“ steht `ByValue` und `ByPropertyName`. Dies bedeutet, dass der Name sowohl das ganze Objekt in der Pipeline sein darf als auch Teil eines Objekts.

Im Fall von

```
"BITS" | Get-Service
```

ist der Pipeline-Inhalt eine Zeichenkette (ein Objekt vom Typ `String`), die als Ganzes auf `Name` abgebildet werden kann.

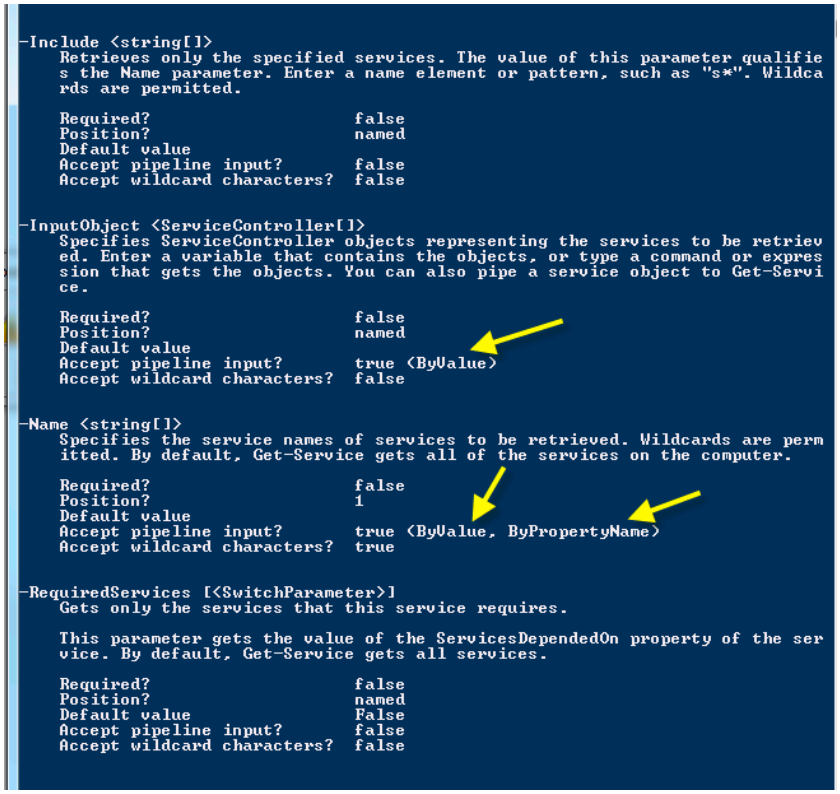



**Bild 5.3** Hilfe zu den Parametern des Commandlets Get-Service

Es funktioniert aber auch folgender Befehl, der alle Dienste ermittelt, deren Name genauso lautet wie der Name eines laufenden Prozesses:

```
Get-Process | Get-Service -ea silentlycontinue | ft name
```

Dies funktioniert über die zweite Option (`ByPropertyName`), denn `Get-Process` liefert Objekte des Typs `Process`, die ein Attribut namens `Name` haben. Der Parameter `Name` von `Get-Service` wird auf dieses `Name`-Attribut abgebildet.

Beim Parameter `-InputObject` ist hingegen nur „ByValue“ angegeben. Hier erwartet `Get-Service` gerne Instanzen der Klasse `ServiceController`. Es gibt aber keine Objekte, die ein Attribut namens `InputObject` haben, in dem dann `ServiceController`-Objekte stecken.

Zahlreiche Commandlets besitzen einen Parameter `-InputObject`, insbesondere die allgemeinen Verarbeitungs-Commandlets wie `Where-Object`, `Select-Object` und `Measure-Object`, die Sie im nächsten Kapitel kennenlernen werden. Der Name `-InputObject` ist eine Konvention.

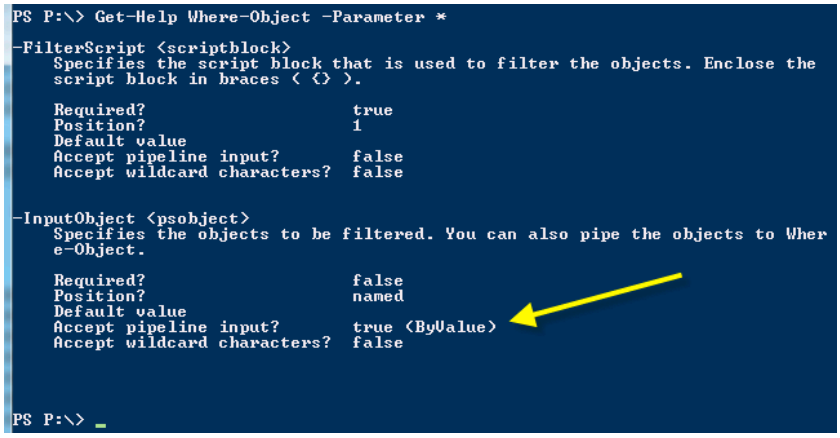



**Bild 5.4** Parameter des Commandlets Where-Object

Leider geht es nicht bei allen Commandlets so einfach mit der Parameterübergabe. Man nehme zum Beispiel das Commandlet Test-Connection, dass prüft, ob ein Computer per Ping erreichbar ist.

Der normale Aufruf mit Parameter ist:

```
Test-Connection  -computername Server123
```

oder ohne benannten Parameter

```
Test-Connection Server123
```

Nun könnte man auf die Idee kommen, hier den Computernamen genau so zu übergeben, wie den Namen bei Get-Service. Allerdings liefert `"Server123" | Test-Connection` den Fehler: *„The input object cannot be bound to any parameters for the command either because the command does not take pipeline input or the input and its properties do not match any of the parameters that take pipeline input.“*.

Warum das nicht geht, kann man in der Hilfe zum Parameter ComputerName des Commandlets Test-Connection erkennen. Dort steht, dass ComputerName nur als „ByPropertyName“ akzeptiert wird und nicht wie beim Parameter Name beim Commandlet Get-Service auch „ByValue“. Das bedeutet also, dass man erst ein Objekt mit der Eigenschaft ComputerName konstruieren und dann übergeben muss:

```
New-Object psobject -Property @{Computername="Server123"} | Test-Connection
```

Das funktioniert zwar, ist aber hässlich und umständlich. Warum Test-Connection und einige andere Commandlets die Eingaben nicht „ByValue“ unterstützen, wusste übrigens das PowerShell-Entwicklungsteam auf Nachfrage auch nicht zu beantworten. Die Schuld liegt hier vermutlich bei dem einzelnen Entwickler bei Microsoft, der die Commandlets implementiert hat.

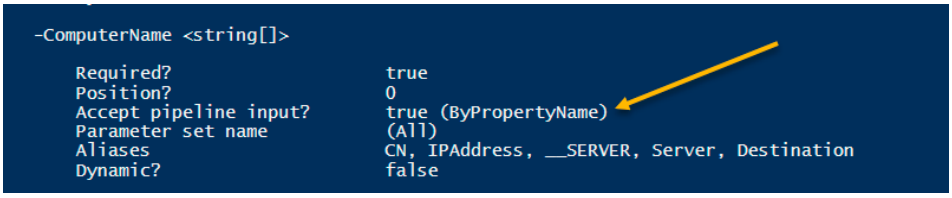



**Bild 5.5** Hilfe zum Parameter ComputerName des Commandlets Test-Connection

## 5.5 Pipelining von klassischen Befehlen

Grundsätzlich dürfen auch klassische Kommandozeilenanwendungen in der PowerShell verwendet werden. Wenn man einen Befehl wie *netstat.exe* oder *ping.exe* ausführt, dann legen diese eine Menge von Zeichenketten in die Pipeline: Jede Ausgabezeile ist eine Zeichenkette.

Diese Zeichenketten kann man sehr gut mit dem Commandlet `Select-String` auswerten. `Select-String` lässt nur diejenigen Zeilen die Pipeline passieren, die auf den angegebenen regulären Ausdruck zutreffen.

In dem folgenden Beispiel werden nur diejenigen Zeilen der Ausgabe von *netstat.exe* gefiltert, die ein großes „E“ gefolgt von zwei Ziffern enthalten.

**TIPP:** Die Syntax der regulären Ausdrücke in .NET wird in Kapitel 7 *„PowerShell-Skriptsprache“* noch etwas näher beschrieben werden.

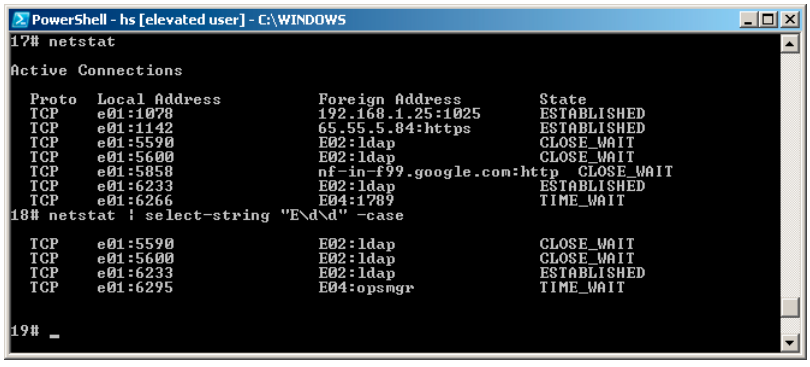



**Bild 5.6** Einsatz von Select-String zur Filterung von Ausgaben klassischer Kommandozeilenwerkzeuge

Ein weiteres Beispiel ist das Filtern der Ausgaben von `ipconfig.exe`. Der nachfolgende Befehl liefert nur die Zeilen zum Thema IPV4:

```
ipconfig.exe /all | select-string IPV4
```

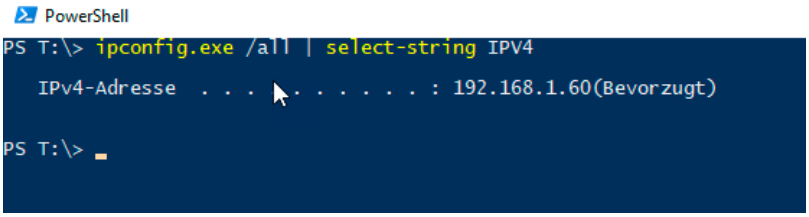



**Bild 5.7** Ausführung des obigen Befehls

## 5.6 Anzahl der Objekte in der Pipeline

Die meisten Commandlets legen ganze Mengen von Objekten in die Pipeline (z. B. `Get-Process` eine Liste der Prozesse und `Get-Service` eine Liste der Dienste). Einige Commandlets legen aber nur einzelne Objekte in die Pipeline. Ein Beispiel dafür ist `Get-Date`, das ein einziges Objekt des Typs `System.DateTime` in die Pipeline legt. Es kann aber auch sein, dass ein Commandlet, das normalerweise eine Liste von Objekten liefert, im konkreten Fall nur ein einzelnes Objekt liefert (z. B. `Get-Process idle`). In diesem Fall liefert die PowerShell dem Benutzer nicht eine Liste mit einem Objekt, sondern direkt das ausgepackte Objekt.

Bis Version 2.0 war es so, dass man eine Liste durch Zugriff auf `Count` oder `Length` nach der Anzahl der Elemente fragen konnte, nicht aber ein einzelnes Objekt.

Das war also erlaubt:

```
(Get-Process).count
```

Das führte aber zu keinem Ergebnis:

```
(Get-Process idle).count
(Get-Date).count
```

Seit PowerShell-Version 3.0 ist dieser Unterschied aufgehoben, man kann immer `Count` und `Length` abfragen und die PowerShell liefert dann eben bei Einzelobjekten eine „1“ zurück. Allerdings schlägt die Eingabehilfe der PowerShell-Konsole und der PowerShell ISE weiterhin weder `Count` noch `Length` als Möglichkeit vor!

**Praxisbeispiel:** Wie viele Prozesse gibt es, die mehr als 20 MB Speicher verbrauchen?

```
(Get-Process | where-object { $_.WorkingSet64 -gt 20mb }).Count
```

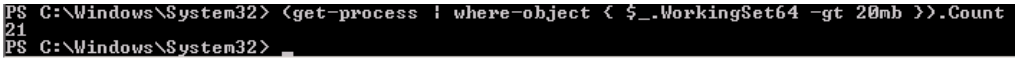



**Bild 5.8** Aufruf von Count für eine Pipeline

## 5.7 Zeilenumbrüche in Pipelines

Wenn sich ein Pipeline-Befehl über mehrere Zeilen erstrecken soll, kann man dies auf mehrere Weisen bewerkstelligen:

- Man beendet die Zeile mit einem Pipe-Symbol [|] und drückt **EINGABE**. PowerShell-Standardkonsole und PowerShell-ISE-Konsole erkennen, dass der Befehl noch nicht abgeschlossen ist, und erwarten weitere Eingaben. Die Standardkonsole zeigt dies auch mit >>> an.
- Man kann am Ende einer Zeile mit einem Gravis [`], ASCI-Code 96, bewirken, dass die nächste Zeile mit zum Befehl hinzugerechnet wird (Zeilenumbruch in einem Befehl). Das funktioniert in allen PowerShell-Hosts und auch in PowerShell-Skripten.

```
PS T:\> Get-Process p* | Sort-Object WorkingsSet |
>> Format-Table id,name,WorkingSet

   Id Name            WorkingSet
   -- ----            ----------
10828 powershell       92942336
15340 powershell_ise  220946432
 1804 powershell       83664896
 4040 powershell       76177408


PS T:\> _
```

**Bild 5.9** Zeilenumbruch nach Pipeline-Symbol

## 5.8 Zugriff auf einzelne Objekte aus einer Menge

Ruft man ein Commandlet auf, das ein einzelnes Objekt liefert, hat man direkt dieses Objekt in Händen. Ruft man z. B. `Get-Date` ohne Weiteres auf, werden das aktuelle Datum und die aktuelle Zeit ausgegeben.

Bei einer Objektmenge kann man, wie oben bereits gezeigt, mit `Where-Object` filtern. Es ist aber auch möglich, gezielt einzelne Objekte über ihre Position (Index) in der Pipeline anzusprechen. Die Positionsangabe ist in eckige Klammern zu setzen und die Zählung beginnt bei 0. Der Pipeline-Ausdruck ist in runde Klammern zu setzen.

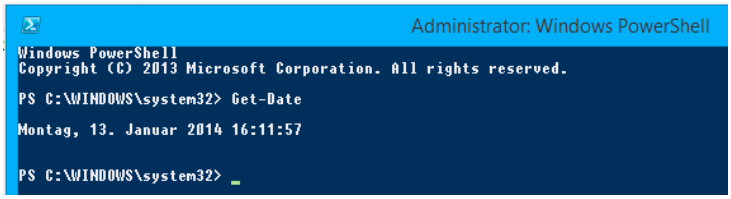



**Bild 5.10** Das aktuelle Datum mit Zeit

**Beispiele:**

Der erste Prozess:

```
(Get-Process)[0]
```

Der dreizehnte Prozess:

```
(Get-Process)[12]
```

Alternativ kann man dies auch mit `Select-Object` unter Verwendung der Parameter `-First` und `-Skip` ausdrücken:

```
(Get-Process i* | Select-Object -first 1).name
(Get-Process i* | Select-Object -skip 12 -first 1).name
```

**HINWEIS:** Während `(Get-Date)[0]` in PowerShell vor Version 3.0 zu einem Fehler führt („Unable to index into an object of type System.DateTime.“), weil `Get-Date` keine Menge liefert, ist der Befehl seit PowerShell-Version 3.0 in Ordnung und liefert das gleiche Ergebnis wie `Get-Date`, da die PowerShell seit Version 3.0 ja aus Benutzersicht ein einzelnes Objekt und eine Menge von Objekten gleich behandelt. `(Get-Date)[1]` liefert dann natürlich kein Ergebnis, weil es kein zweites Objekt in der Pipeline gibt.

Die Positionsangaben kann man natürlich kombinieren mit Bedingungen. So liefert dieser Befehl den dreizehnten Prozess in der Liste der Prozesse, die mehr als 20 MB Hauptspeicher brauchen:

```
(Get-Process | where-object { $_.WorkingSet64 -gt 20mb } )[12]
```

```
PS C:\Windows\System32> (get-process)[0]

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     20       2     1968       2664    17     0,03   2784 cmd

PS C:\Windows\System32> (get-process)[12]
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     69       9     1484       4196    41     0,03   2100 dlpwdnt

PS C:\Windows\System32> (get-process | where-object { $_.WorkingSet64 -gt 20mb } )[12]
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    685      29    53924      59544   291    34,39   4984 powershell

PS C:\Windows\System32>
```

**Bild 5.11** Zugriff auf einzelne Prozessobjekte

## 5.9 Zugriff auf einzelne Werte in einem Objekt

Manchmal möchte man nicht ein komplettes Objekt bzw. eine komplette Objektmenge verarbeiten, sondern nur eine einzelne Eigenschaft.

Oben wurde bereits gezeigt, wie man mit `Format-Table` auf einzelne Eigenschaften zugreifen kann:

```
Get-Process | Format-Table ProcessName, WorkingSet64
```

Hat man nur ein einzelnes Objekt in Händen, geht das ebenfalls:

```
(Get-Process)[0] | Format-Table ProcessName, WorkingSet64
```

`Format-Table` liefert aber immer eine bestimmte Ausgabe, eben in Tabellenform mit Kopfzeile. Wenn man wirklich nur einen bestimmten Inhalt einer Eigenschaft eines Objekts haben möchte, so verwendet man die in objektorientierten Sprachen übliche Punktnotation, d. h., man trennt das Objekt und die abzurufende Eigenschaft durch einen Punkt (Punktnotation).

**Beispiele:**

```
(Get-Process)[0].Processname
```

Die Ausgabe ist eine einzelne Zeichenkette mit dem Namen des Prozesses.

```
(Get-Process)[0].WorkingSet64
```

Die Ausgabe ist eine einzelne Zahl mit der Speichernutzung des Prozesses.

Mit den Einzelwerten kann man weiterrechnen, z. B. errechnet man so die Speichernutzung in Megabyte:

```
(Get-Process)[0].WorkingSet64 / 1MB
```

```
PS C:\Windows\System32> (get-process)[0] | Format-Table ProcessName, WorkingSet64

ProcessName                                                          WorkingSet64
-----------                                                          ------------
cmd                                                                       2727936


PS C:\Windows\System32> (get-process)[0].Processname
cmd
PS C:\Windows\System32> (get-process)[0].WorkingSet64
2727936
PS C:\Windows\System32> (get-process)[0].WorkingSet64 / 1MB
2,6015625
PS C:\Windows\System32>
```

**Bild 5.12** Ausgabe zu den obigen Beispielen

Weitere Anwendungsfälle seien am Beispiel `Get-Date` gezeigt. `Year`, `Day`, `Month`, `Hour` und `Minute` sind einige der zahlreichen Eigenschaften der Klasse `DateTime`, die `Get-Date` liefert.